NEAR-DATA PROCESSING FOR DYNAMIC GRAPH ANALYTICS

A Dissertation
Presented to
The Academic Faculty

By

Eric Robert Hein

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

August 2018

NEAR-DATA PROCESSING FOR DYNAMIC GRAPH ANALYTICS

Approved by:

Professor Tom Conte, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor David Bader
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Sudhakar Yalamanchili, Committee Chair
School of Electrical and Computer Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Professor Milos Prvulovic
School of Computer Science
Georgia Institute of Technology

Date Approved: May 15, 2018

Of making many books there is no end, and much study wearies the body.
Now all has been heard;
here is the conclusion of the matter:
Fear God and keep his commandments,
for this is the duty of all mankind.
For God will bring every deed into judgment,
including every hidden thing,
whether it is good or evil.
Ecclesiastes 12:12b-14

ACKNOWLEDGEMENTS

Thanks to my parents, Robert and Carolyn Hein, for their life-long love and support.

Thanks to the many excellent teachers and professors at Apostles Lutheran School and Luther Preparatory School, for providing me with a world-class Christian education that prepared me for the rigors of college and post-graduate study.

Thanks to Eric Durant, for encouraging me to pursue a PhD in the first place.

Thanks to my fellow TINKER group members, especially Brian Railing and Jesse Beu, for involving me in their research and teaching me how to be a grad student.

Thanks to my labmate, boss, mentor, and friend, Jason Poovey. By introducing me to TINKER, STINGER, and GTRI, you connected me to exciting projects and amazing people, and played an essential role in shaping my career at Georgia Tech.

Thanks to the folks in the "Innovative Computing" group at GTRI, for keeping me sane with lunch discussions, nerf guns, foosball, snakes, spiders, and scorpions.

Thanks to my advisor, Tom Conte, for teaching me how to defend my work and giving me the freedom to go down rabbit holes.

Thanks to the committee members, Sudhakar Yalamanchili, David Bader, Richard Vuduc, and Milos Prvulovic, for their time in reviewing and evaluating my work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SOURCE CODES

SUMMARY

Massive data rates in cybersecurity, simulation, and social media analysis applications are driving rapid advances in the field of streaming graph analytics. The data structures that enable streaming graph analytics pose unique challenges for high-performance computing system designers. When the sorted, contiguous arrays of static graphs are replaced with the fragmented, linked data structures of dynamic graphs, these systems struggle to reach the memory bandwidth saturation point. Behaviors such as pointer-chasing and poor spatial locality expose the true latency of modern memory devices, which has not kept up with processor clock rates.

This dissertation develops a streaming graph benchmark, DynoGraph, which is distinguished from static graph benchmarks by the use of realistic streaming graph inputs and dynamic graph data structures. The benchmark is used to expose performance pitfalls in existing implementations. These insights flow into the design of near-memory accelerators for streaming graph analytics, as well as software improvements. The Emu architecture is identified as a promising solution for accelerating algorithms with low spatial locality, unbalanced parallelism, and fine-grained memory accesses, since it is able to maintain high memory bandwidth utilization in a worst-case pointer-chasing scenario. The work culminates in a characterization of the Emu Chick hardware prototype, proposing efficient programming primitives, highlighting necessary system improvements, and demonstrating the potential for greatly improved performance on this important class of workloads.

CHAPTER 1
INTRODUCTION

1.1 Motivation

In the present era of Big Data, massive amounts of digital information are created every second. Computer-driven analytical systems that can derive understanding from this data in order to discover trends and take action quickly are driving progress in commerce, politics, and science. Of particular interest is the field of streaming graph analytics, in which events are gathered to form and analyze a web of relationships in real time. The successful implementation of these algorithms on emerging high-performance computing systems demands innovation throughout the entire computing stack, from hardware to software.

For decades, researchers have known about the "memory wall" [3, 4], a point where overall computing performance would be hamstrung by the memory system. While CPU performance has experienced exponential growth in keeping with Moore's Law, main memory performance has struggled to keep up. Architects have done a remarkable job of hiding this problem by increasing the depth of the memory hierarchy and the minimum width of each access.

Computer architecture research has long relied on benchmarks to measure the effect of innovation and drive product development. Suites like SPEC [6], PARSEC [7], and SPLASH-2 [8] were developed to model typical applications for chip multiprocessors (CMP's), and are still frequently used to measure the effectiveness of new improvements to the microarchitecture. As a result, computer architecture research has catered to the needs of scientific computing. Modern processors and memory systems are designed to maximize FLOPS for applications with spatial locality and high rates of data reuse, placing three (or even four [9]) levels of cache memory between the processor and the memory system. Such

Table 1.1: True memory latency for successive generations of DRAM technology. Reproduced from [5].

Technology	Module Speed (MT/s)	Clock Cycle Time (ns)	CAS Latency (# of clock cycles)	True Latency (ns)
SDR	100	8.00	3	24.00
SDR	133	7.50	3	22.50
DDR	335	6.00	2.5	15.00
DDR	400	5.00	3	15.00
DDR2	667	3.00	5	15.00
DDR2	800	2.50	6	15.00
DDR3	1333	1.50	9	13.50
DDR3	1600	1.25	11	13.75
DDR4	1866	1.07	13	13.93
DDR4	2133	0.94	15	14.06
DDR4	2400	0.83	17	14.17
DDR4	2666	0.75	18	13.50

a memory hierarchy can deliver hundreds of gigabytes per second of bandwidth and massively reduced latency for workloads that exhibit temporal and spatial locality. Although main memory latency has remained relatively constant in recent years (as documented in Table 1.1), this has been masked by increasing levels of memory parallelism and bus transfer rate. Processors mitigate memory latency with out-of-order execution and hardware prefetchers.

But the algorithms that find relationships in massive data streams, such as the analysis of network packet data [10] or social media posts [11] in real time, are not so well-behaved. Streaming graph analytics applications stream through large amounts of memory in data-dependent order, and perform a small number of computations for each byte loaded [12]. As a result, they inefficiently utilize cache space, main memory bandwidth, and functional units in the processor. They do not generate enough pending requests to saturate main memory bandwidth [13], due to cache hits and limited reorder window size [14]. Graph applications require a high degree of communication between processing elements, which limits scaling in multi-socket and multi-node systems. Furthermore, the irregular structure of most real-world graphs [15] makes it difficult to evenly partition work and data

across distributed resources and parallel execution units. The unpredictable access pattern of graph algorithms disrupts the branch predictor [16] and the hardware prefetcher.

The recent explosion of interest in high-performance graph processing has led to the rapid development of specialized algorithms [17, 18, 19, 20] and systems [21]. Graphs place heavy stress on existing memory hierarchies, driving research into novel memory technologies such as the Hybrid Memory Cube (HMC) [22], High Bandwidth Memory (HBM) [23], and other novel memory technologies [24], in addition to near-memory accelerators that make more efficient use of this bandwidth [25, 26, 27, 28]. Accurately evaluating the effectiveness of these and other innovations will require realistic graph benchmarks based on applications that have been deployed in the real world. The vast diversity of graph data sets and algorithms with respect to other application domains [29] makes finding a representative set of proxy applications even more critical.

Traditionally, the most important metric associated with a memory system has been peak memory bandwidth, which measures the maximum number of bytes that can be transferred to the processor per unit time under ideal conditions. Algorithms that hit this "speed limit" will surely benefit from the increased performance that these technologies deliver. But simply increasing peak memory transfer rate will be of little benefit to irregular applications such as streaming graph analytics, especially if these systems continue to rely on assumptions of locality and predictable access patterns. Latency-bound algorithms depend not on the peak rate of data transfer, but on the time for a single memory request to complete.

Since the latency of main memory is fixed, accelerating the traversal of large linked data structures such as binary trees, linked lists, and hash maps can only be achieved by a memory system that efficiently generates and satisfies a large number of parallel requests at the granularity of a single pointer, without relying on locality of reference. Examples of such architectures include the Cray XMT [30], GoblinCore-64 [31], and most recently Emu [32]. Emu's notion of migratory memory-side processing is designed for low-locality

situations, and promises to deliver impressive speed-ups for graph analytics applications.

1.2 Contributions

This dissertation will show that **emerging near-memory accelerators and memory-centric architectures must be evaluated and co-designed with a realistic streaming graph analytics benchmark.** This work makes the following contributions towards the design and implementation of a memory-centric architecture that is optimized for the rapid construction, modification, and analysis of massive streaming graph datasets:

- Chapter 2 presents DynoGraph, a benchmark suite for streaming graph analytics. DynoGraph will be a necessary tool for evaluating the performance of emerging near-memory architectures for streaming graph analytics.
  - DynoGraph provides real-world streaming graph inputs, which contain duplicates and bursts of updates to high-degree vertices.
  - DynoGraph measures the performance of incremental graph construction, allowing a fair evaluation of write-optimized data structures in competition with read-optimized data structures such as CSR.
  - DynoGraph forces graph algorithms to run on a "mature" graph data structure that has been constructed organically, as opposed to an "optimal" memory layout that has been directly loaded from an on-disk snapshot.
- Chapter 3 presents several proposals relating to acceleration of the STINGER streaming graph engine. These explorations highlight the most difficult problems in streaming graph analytics, and suggest directions for exploration in future hardware development.
  - An improved parallel algorithm for applying a batch of updates to a STINGER graph in an efficient manner, which reduces duplicated work between multiple

threads.

  - The design and simulation of a near-memory, content-directed fetch unit to accelerate STINGER edge list traversals.
  - An exploration of the scalability of fragmented edge list traversal in a near-memory accelerator with access to high-bandwidth, multi-channel memory.

- Chapter 4 presents a characterization of the Emu Chick hardware prototype. These results provide insight into the applicability of the Emu architecture to sparse data sets and memory-centric problems. In particular, they showcase the unique ability of the Emu Chick to maintain high memory bandwidth utilization in the presence of low spatial locality.

  - Techniques for achieving memory bandwidth scalability on the STREAM benchmark.
  - A characterization of data-dependent random-access memory behavior on the "Pointer Chasing" benchmark.
  - Simulation results indicating performance scalability past the initial hardware prototype.

- Chapter 5 presents the first implementation of a streaming graph analytics engine on the Emu Chick hardware prototype. This work demonstrates the feasibility of streaming graph analytics on a novel memory-centric architecture. Furthermore, it suggests improvements to apply to the Emu architecture as well as new directions for research into processing-near-memory for streaming graph analytics.

  - A distributed, streaming graph data structure, using a memory layout and parallel primitives optimized the Emu architecture.
  - Graph algorithms running on the Emu Chick, including a modified implementation of breadth-first search that takes advantage of fine-grained remote writes.

- Discussion of best practices to employ and performance pitfalls to avoid when programming for the Emu Chick.

Finally, Chapter 6 summarizes the impact of these contributions and suggests directions for future work.

CHAPTER 2
DYNOGRAPH

2.1 Motivation

Prior work on large graph characterization has focused on *static graphs* – graphs that are loaded from a static snapshot file (Figure 2.1, top). Ingesting graphs in one pass allows for use of compact data structures such as Compressed Sparse Row (CSR), and enables optimizations such as sorting by degree to increase locality.

But many real-world graphs are not built in one pass. Rather, graphs evolve over time as data is added and removed (Figure 2.1, bottom). Prior benchmark studies, although useful for capturing graph analytic execution behaviors in certain situations, fail to capture key performance characteristics. Applications such as streaming analysis of social media [33] and network security data [34] require a dynamic graph workflow, which intensifies the already harsh computational demands of static graph analytics and breaks many of the simplifying assumptions that enable performance. Figure 2.2 plots the total time taken to load 100 incremental graph snapshots and compute the PageRank on each one. The bar cluster on the left uses a read-optimized graph data structure. The middle and right bar clusters use a write-optimized graph data structure. The rightmost bar cluster loads the graph snapshots incrementally in fixed-size batches, as would occur in a dynamic graph workflow.

2.2 Related Work

2.2.1 Shared-memory Graph Benchmarks

GraphBIG [35] implements a graph benchmark suite using graph data structures based on IBM System G [36]. It includes benchmarks that operate on not only the graph structure,

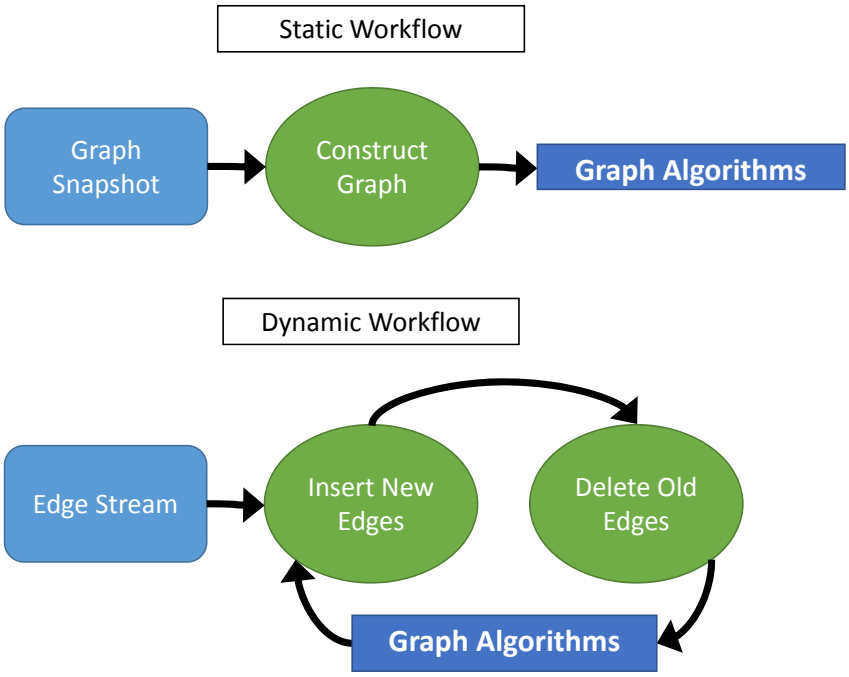


Figure 2.1: Unlike the static graph workflow, which loads a static graph snapshot, the dynamic graph workflow processes a continuous stream of edges.

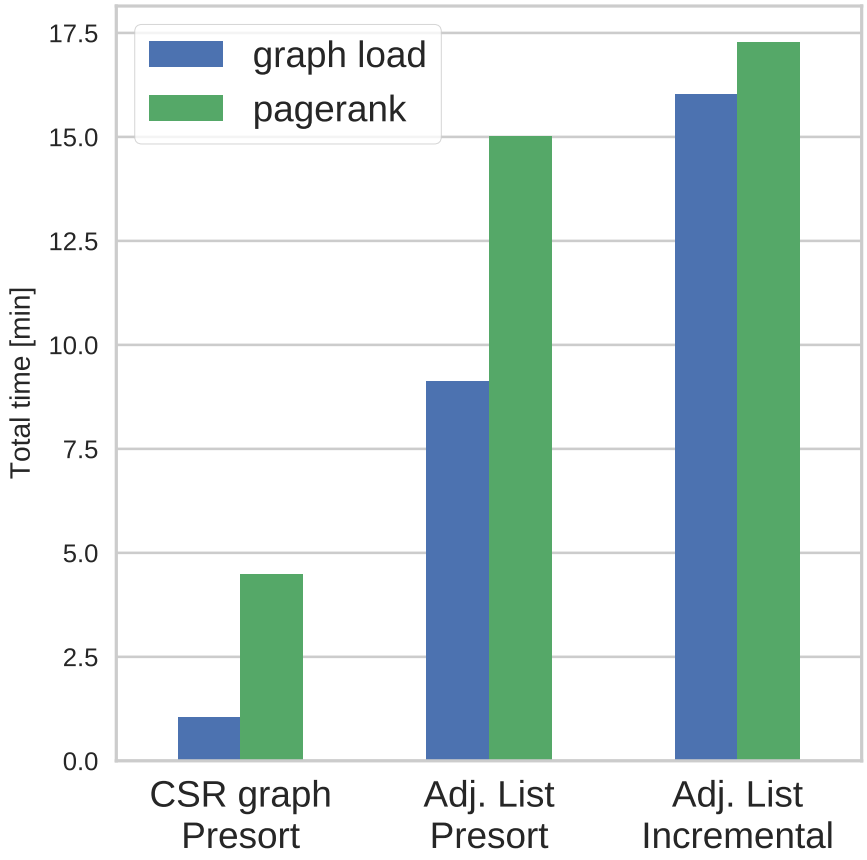


Figure 2.2: Total time taken to load 100 graph snapshots and compute the PageRank on each one. The bar cluster on the left uses a read-optimized graph data structure. The middle and right bar clusters use a write-optimized graph data structure. The rightmost bar cluster loads the graph snapshots incrementally in fixed-size batches.

but also graph properties, and additionally provides three dynamic graph computation kernels for graph construction, update, and transform. Unlike DynoGraph, these kernels do not affect the graph structure for other graph algorithms.

There are several open source graph benchmarks that target shared-memory systems, including Lonestar [37], CRONO [38], Ligra [39, 40], and GAP [41]. None of these benchmarks implement incremental graph construction, though GAP was modified in section 2.6 to work with the DynoGraph driver program by recreating a static graph data structure in each batch. A similar technique could be extended to each of these benchmarks.

Both the Stanford SNAP project [42] and the Koblenz Network collection (KONECT) [43] provide a wide range of graph datasets. Some of the KONECT datasets even include edge arrival times, but they do not contain duplicate edges.

The Boost C++ libraries include a highly flexible set of templates for graph traversal. GraphTool [44] adds OpenMP pragmas for shared-memory parallelization, and additionally allows using these primitives from Python.

2.2.2 Distributed Graph Frameworks

Graph500 [45] is a benchmark for large-scale graph processing. It consists of two timed kernels, one to construct a graph representation from an edge list, and another to measure edge traversal rate during a series of breadth-first searches. Unlike DynoGraph, Graph500 does not require implementations to update an existing graph, nor does it require traversal of a graph that has been incrementally constructed.

Graphalytics [46] compares the performance of several distributed graph frameworks, including Giraph, GraphX [47], MapReduce, and neo4j [48], finding that low locality of reference, uneven load balancing, and poor scalability are common bottlenecks. However, Kineograph [49] achieved timely distributed streaming graph processing, and STAPL [50] is designed for both shared and distributed memory systems. The CloudSuite [51] benchmark recently added in-memory graph analytics using Apache Spark. GraphBench [52]

hosts several common graph algorithms implemented in various distributed graph frameworks, along with datasets for benchmarking.

The parallel Boost Graph Library (PBGL) [53] is an in-memory graph analytic library distributed with the Boost C++ framework and parallelized using MPI. A preliminary PBGL implementation of DynoGraph was found to be orders of magnitude slower than the other engines evaluated, even when running on a single node. This was due to the lack of dynamic graph partitioning and the extra overhead required to communicate between ranks. Because of these difficulties, the scope of this work was limited to shared-memory graph processing implementations.

2.3 Dynamic Graph Data Structures

A dynamic graph data structure must be engineered to allow for efficient modification of edge and vertex data. This usually means representing the list of edges as a linked data structure rather than a contiguous array. Compressed-Sparse-Row (CSR) is an efficient, array-based representation for static graphs, but it is impossible to update without moving a large portion of the graph in memory (left half of Figure 2.3). It is not feasible for a dynamic graph data structure to allocate a fixed quantity of storage for each vertex's edges. If the graph is sparse and follows a power law distribution, this will be wasteful for most vertices. On the other hand, a few vertices may be connected to the entire graph. Since any vertex could become highly connected at any time, the data structure must dynamically allocate memory for edge storage, and maintain a chain of pointers to valid edge data.

The traversal of dynamic graph data structures introduces new problems for the processor and memory system. Figure 2.3 gives an example of how the layout of a graph in memory can become fragmented as edges are inserted and deleted over time. In contrast with CSR, this means that walking the list of edges for a vertex will not generate memory accesses with a predictable stride pattern, but instead will jump randomly around memory. These pointer dereferences limit memory parallelism and stall the processor pipeline. The

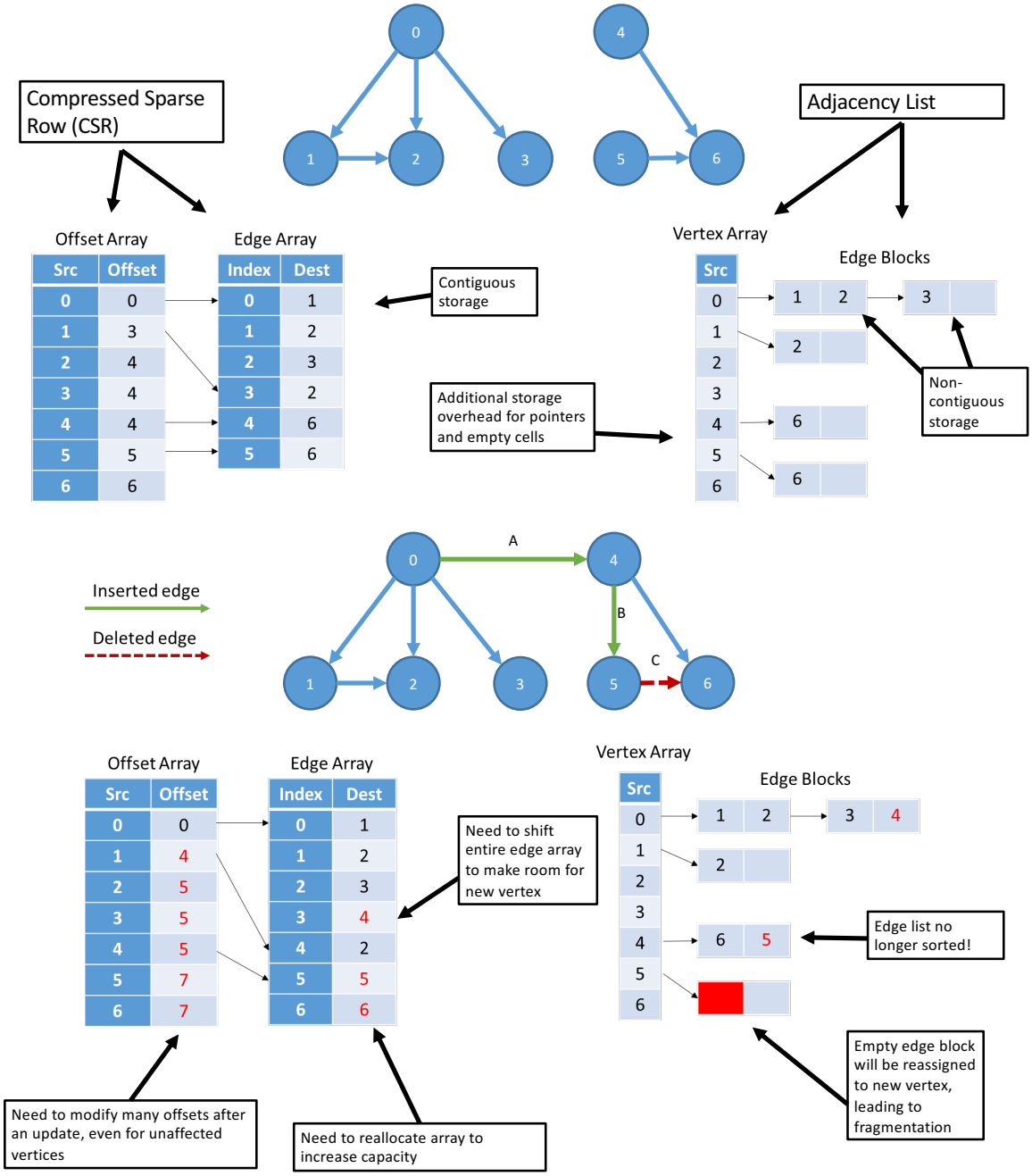


Figure 2.3: The way a graph is stored in memory affects the performance of graph analytics. The Compressed-Sparse-Row (CSR) format stores graphs more efficiently, but an adjacency list is easier to update.

cost of pointer-chasing can be amortized by storing several edges in each linked-list node, but this requires keeping additional metadata about which edges are valid. These tracking structures increase the storage capacity requirements, reduce the percentage of fetched data that are actually useful, and introduce irregularity into the traversal code.

When the structure of the graph changes, all metrics computed on that graph are invalidated and must be recalculated. The introduction of even a single edge can connect two previously unconnected sub-graphs, changing the shortest-path information for the majority of vertices. However in many cases, an algorithm can save time by examining which vertices are actually affected by an update and limit the scope of computation to only this fraction of the graph [11]. Another strategy is to employ an approximate algorithm that uses graph updates to refine an estimate of the desired metric [18], or sampling of the edge stream to reduce the size of the graph that must be stored [54].

A dynamic graph benchmark suite should measure the performance of edge updates and deletions in addition to timing the graph algorithms. This forces the use of dynamic data structures and captures the trade-off between efficient graph traversal and efficient graph update. The way the graph is constructed is critical. Loading a snapshot from disk does not create the same in-memory layout compared with incrementally performing edge insertions and deletions. The graph algorithms must be benchmarked after the edge stream has occurred, in order to truly emulate the fragmented in-memory layout of a real dynamic graph application. To meet this goal, the input to a dynamic graph benchmark must be a stream of edge updates as they would occur in time. This technique introduces two important aspects of realism; first, edges will not be sorted by degree or by source vertex, forcing more irregularity during graph construction. Second, there will be duplicate edge updates that must be merged during the edge stream process.

2.4 Specification

Like many other benchmark suites [41, 45, 55], DynoGraph is specified in terms of high-level requirements instead of exclusively providing source code for a single implementation. This allows adopters the freedom to optimize the performance of DynoGraph using their graph processing framework of choice, and opens the door to high-performance hardware/software co-design techniques.

An implementation of the DynoGraph benchmark consists of a graph processing engine (software) running on a graph processing system (hardware). A driver program instructs the engine to execute a list of steps. The engine persists graph state between steps. Each step is timed, and corresponding step times form a point of comparison across engines and systems.

There are three kinds of steps: insertions, deletions, and algorithms.

Insertions The driver program provides the engine with a list (batch) of directed edges from the input edge list. Each edge consists of four 64-bit integers, labeled as (*source*, *destination*, *weight*, *timestamp*). The list may contain duplicates, but is guaranteed to be sorted in ascending order with respect to timestamp. Each directed edge that does not already exist in the graph must be inserted into the graph. Edges that already exist in the graph (i.e. there is already an edge from source to destination) must update the existing edge by adding the weights and overwriting the timestamp.

Deletions The engine removes all edges with a timestamp older than a threshold value, which is derived from the window size in each batch. The driver program increases the threshold value as the benchmark proceeds, effectively creating a sliding window over the edge list. The window size is a benchmark parameter, specified as a percentage of the total time span of the edge list. For example, if the first edge in the list occurred at 8:00am, and the last edge occurred at 5:00pm on the same day, then a window size of 20% would specify deletion of all edges that are more than 2 hours

epoch	batch	time	threshold	steps
0	0	8:00am	>= 8:00am	deletions, insertions
	1	9:00am	>= 8:00am	deletions, insertions
	2	10:00am	>= 8:00am	deletions, insertions
	3	11:00am	>= 9:00am	deletions, insertions
	4	12:00pm	>= 10:00am	deletions, insertions
1	5	1:00pm	>= 11:00am	deletions, insertions, bfs, cc, pagerank
	6	2:00pm	>= 12:00pm	deletions, insertions
	7	3:00pm	>= 1:00pm	deletions, insertions
	8	4:00pm	>= 2:00pm	deletions, insertions
	9	5:00pm	>= 3:00pm	deletions, insertions, bfs, cc, pagerank

Figure 2.4: Example of a small DynoGraph benchmark. There are two epochs, each with 5 batches of insertions and deletions. The window size for deletions is set to 20% of the total time span of the graph, or 2 hours. Graph algorithms run at the end of each epoch.

old.

Algorithms Graph algorithms run against the current state of the graph. Dynamic algorithms are permitted to persist results between steps, and are provided with a list of the insertions and deletions that occurred since the last time the algorithm ran. Alternately, static algorithms may simply recompute from scratch in each step. The algorithms selected for this work are Breadth-First Search (bfs), Connected Components (cc)[56], and PageRank (pagerank)[57].

Steps are further organized into epochs. Each epoch consists of several fixed-size batches of insertions alternating with deletions, ending with a single algorithm step. Figure 2.4 shows an example of how steps are grouped into epochs. The total number of edge insertions per epoch is determined in the following manner: First, divide the length of the edge list by a fixed batch size. Then, insertion steps are evenly divided into epochs. There must be at least one batch in each epoch, and at least one edge in each batch. Thus the controlling parameters for DynoGraph are the input edge list, the number of epochs, the batch size, and the window size. Together, these parameters completely specify the sequence of steps that must be run as well as the state of the graph at the beginning of each step.

Table 2.1: DynoGraph input graph sizes

	Description	# of Vertices	Total Edges	Unique Edges	Edge Factor
sc15	NetFlow data from SCinet 2015	8 M	270 M	38 M	4.73
dns2	Passive DNS	75 M	236 M	103 M	1.40
worldcup	Twitter data from the 2014 World Cup	14 M	63 M	35 M	2.58
RMAT	RMAT scale 24	17 M	266 M	266 M	16

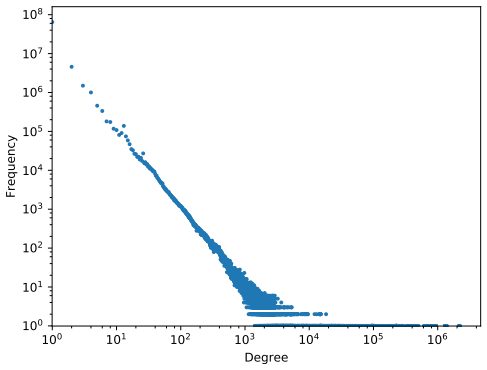
2.5 Inputs

DynoGraph introduces three new streaming datasets, which are distinguished from static graph datasets by several important properties. Each edge between two vertices represents an interaction between two real-world entities at a moment in time. Consequently, the edge list is sorted with respect to time only and there are many duplicate edges connecting the same source and destination vertices. The degree of duplication varies on a per-batch basis; a given batch may consist entirely of new edges, entirely of updates to existing edges, or any mixture of the two.

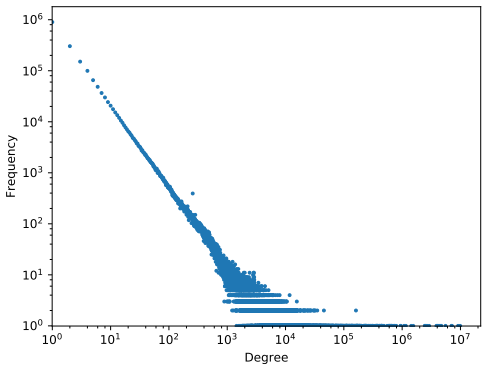
Table 2.1 shows the number of vertices and edges in each edge list. In general, the DynoGraph datasets have tens of millions of vertices and hundreds of millions of edges. Figure 2.5 shows the degree distribution of each graph in the final epoch. Each graph has a similarly skewed distribution; there are many vertices with few neighbors, and a few vertices with many neighbors. This property makes it difficult to evenly partition work among threads, as the amount of work per vertex may differ by orders of magnitude. DynoGraph edge streams contain many duplicate edges. Edges may be duplicated within a batch, or they may be already present in the graph. Figure 2.6 shows the number of unique edges in each batch.

SC 2015 NetFlow [58] This dataset was collected at the SC15 conference. A team of researchers collected NetFlow data from SCinet for the duration of the conference, and ran real-time analytics over the resulting graph. In this graph, each vertex represents

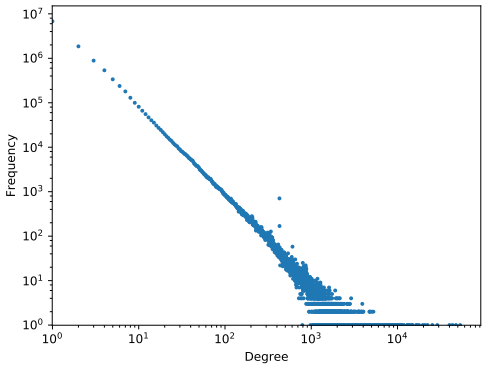
Figure 2.5: Degree distributions of the graphs tested in this work.



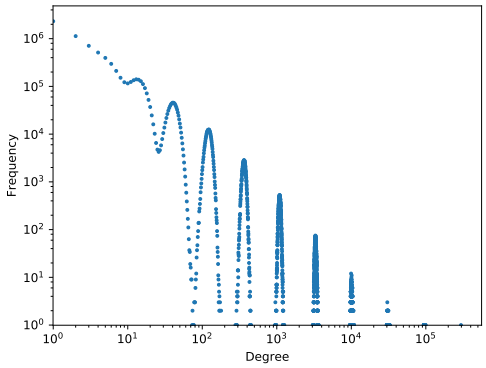
(a) dns2



(b) sc15



(c) worldcup



(d) rmat

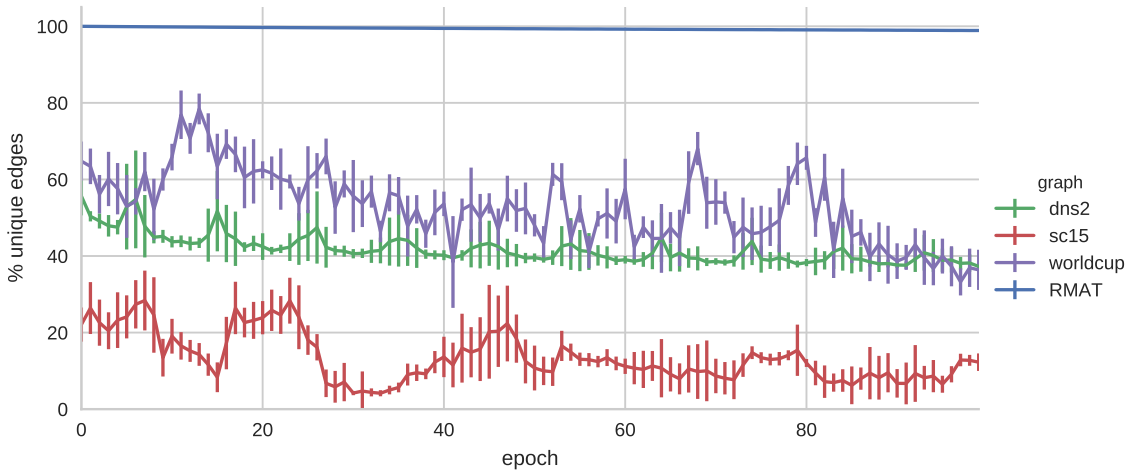


Figure 2.6: Plots the mean rate of edge duplication for DynoGraph datasets in each epoch. Error bars show the standard deviation from the mean, computed over all batches in the epoch.

an IP address. An edge represents that data was transferred between those two hosts. Edges are weighted with the number of bytes transferred. This dataset is representative of the types of graphs generated when analyzing real networks for cybersecurity threats where the application of PageRank, betweenness centrality, and community detection are used to find bot-nets, emergent graph behavior, and potential distributed denial of service (DDoS) attack targets.

Passive DNS [59] This dataset is an anonymized graph of real DNS data collected over the entire campus network of a large university. Edges in this graph are created by domain name look-ups for a given host, tracking the resolution of the name requests as they traverse the hierarchy of name servers. Real-time applications of this data include using centrality and community detection to locate hackers and bot-net control networks, which maliciously manipulate DNS records to accomplish their goals.

Twitter Social media remains one of the most prevalent applications of dynamic graph analysis. This data set represents the Twitter graph of mentions between Twitter users that occurred over a three week period during the 2014 World Cup. The Twitter graph was collected targeting hash tags specific to the World Cup as well as hash tags

specific to the Twitter campaigns of the primary sponsors.

RMAT[60] A synthetic graph has also been included for comparison. The parameters were chosen to match the specification in the Graph500 benchmark, and the scale was chosen to match the other graphs in this work (A=0.55, B=0.20, C=0.10, D=0.15, edge factor=16, scale=24). Each edge has a fixed weight and the timestamps increment sequentially.

2.6 Implementations

Two high-performance graph engines were profiled using the DynoGraph benchmark. The Graph Algorithm Platform (GAP) Benchmark Suite [41] uses a static CSR graph representation, while STINGER [61] uses a dynamic graph data structure designed to support rapid updates. Both are designed to run on shared-memory multicore machines, utilizing OpenMP [62] and atomic intrinsics to implement threading and synchronization. The contrasting design goals of STINGER and GAP lead to differences in how they construct the graph from an edge list, how they store the graph in memory, and how they iterate through vertices and edges.

STINGER is an in-memory graph data structure that is designed for massive streaming data analytics on shared-memory machines. STINGER uses a contiguous array for vertex storage. Each vertex contains a pointer to a linked list of edge blocks. Each edge block contains a header followed by a fixed-length contiguous array of edges. A major design goal of the STINGER data structure was to support efficient multi-threaded graph updates on the order of millions of edges per second [63]. Graph insertions in STINGER are parallel-safe, allowing multiple graph updates to occur simultaneously, even for the same vertex. More details of the STINGER batch insertion algorithm are discussed in section 3.1.

Several modifications were made to the GAP source code to implement DynoGraph. The size of the vertex identifier was increased to 64 bits and a weight and timestamp field was added to each edge. The existing code was extended to support graph updates in the

following manner:

1. The graph is converted to an edge list.
2. The batch of updates is appended to this list.
3. The edge list is sorted and de-duplicated to yield a list of unique edges.
4. The existing GAP code uses parallel prefix sum to create the offset array, resulting in a finalized CSR graph.

Despite being very costly to update, GAP should outperform dynamic data structures for graph traversal due to its compact and contiguous layout.

2.7 Experimental Methodology

Each benchmark was compiled using gcc 4.9.2 with full optimization enabled (`-O3`) and run on a dual-socket Intel server (Xeon E5-2670 @2.60GHz) with 64GB of DDR3 RAM. The DynoGraph experiments in this section set the batch size to 50,000 edges and ran 100 epochs per benchmark. Each of the three DynoGraph datasets were tested, in addition to the scale-24 RMAT dataset mentioned earlier. The aforementioned configurations were run with both GAP and STINGER to highlight the differences between read-optimized and write-optimized graph layouts.

2.8 Results

The first comparison between STINGER and GAP highlights the trade-offs inherent in supporting efficient insertions and deletions. Recall that GAP uses a CSR graph representation while STINGER uses a linked adjacency list.

Several efforts have been made to ensure that the comparison is between graph data structures and not between the particular implementation of a graph algorithm. The DynoGraph test harness ensures that all the engines choose the same starting vertex for algo-

rithms like BFS. Despite these efforts there are still some incongruities between engines. GAP uses a direction-optimizing BFS [64] that reduces the number of edges examined, while STINGER uses the standard technique of parallelizing over each frontier. Despite these discrepancies, the number of edges traversed in both engines matches closely.

Figure 2.7 compares the insert and update rate of the STINGER and GAP graph engines. The X-axis spans the entire benchmark, with each data point representing the number of seconds required to perform 50,000 updates on the current graph state. The insert time for GAP grows with each batch. Since GAP needs to process the entire edge list for each batch of insertions, the run time is proportional to the number of edges in the graph.

STINGER performs better overall in this category because it can add new edges without moving unaffected ones. However there are several anomalies that must be explained. Fig. 2.9b overlays data onto the insertion time plot that explains this trend. For each batch the maximum degree of any vertex that is updated in that batch is plotted on a secondary axis. Now it is clear that the first spike in sc15 is caused by a large number of new edges being added to a vertex that already has the highest degree in the graph. A thread in STINGER must search each vertex's adjacency list for existing edges before attempting to add new edges to the end of the list. Then, it must contend with other threads to atomically append a new edge block to the end of the list. After the degree of this vertex levels off around batch 1500, the performance improves dramatically. While this vertex continues to be updated in future batches, there are no more new edges, meaning most threads will not need to traverse the entire list to complete their update. A similar phenomenon occurs near batch 3000 in sc15 and in batches 200 through 11000 in worldcup.

Despite having a skewed degree distribution like the other graphs, insertion performance is flat throughout the batches of the RMAT graph dataset. While RMAT does generate high-degree vertices, it seems to spread their neighbors evenly across all batches instead of generating hot spots. This suggests that RMAT datasets are not an effective tool for benchmarking incremental graph construction because they lack the irregularities and

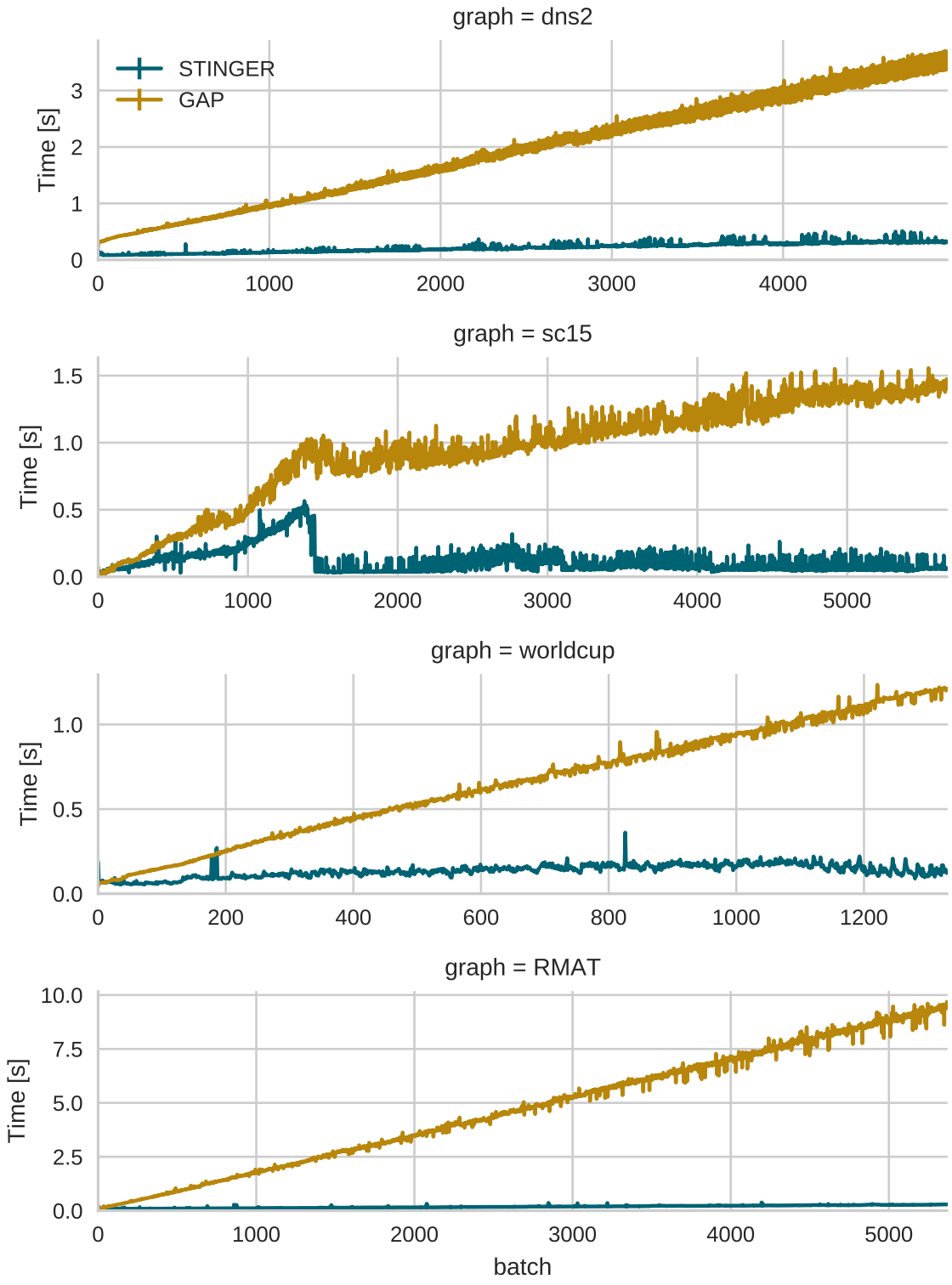


Figure 2.7: Graph update performance comparison between the STINGER and GAP graph engines. GAP's insert time scales with the size of the graph, while STINGER is flat except for several anomalies which can be explained by examining the composition of each batch.

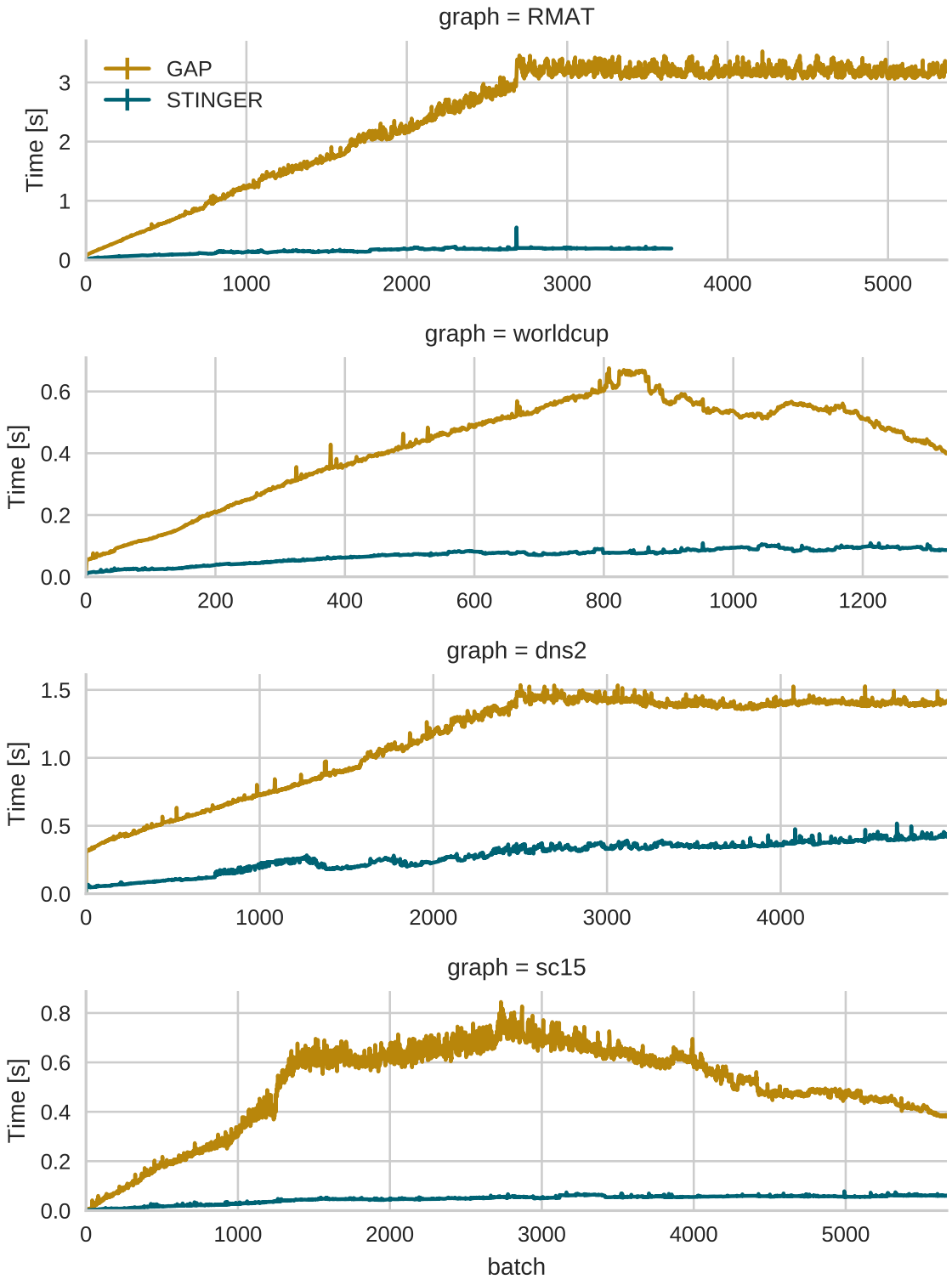
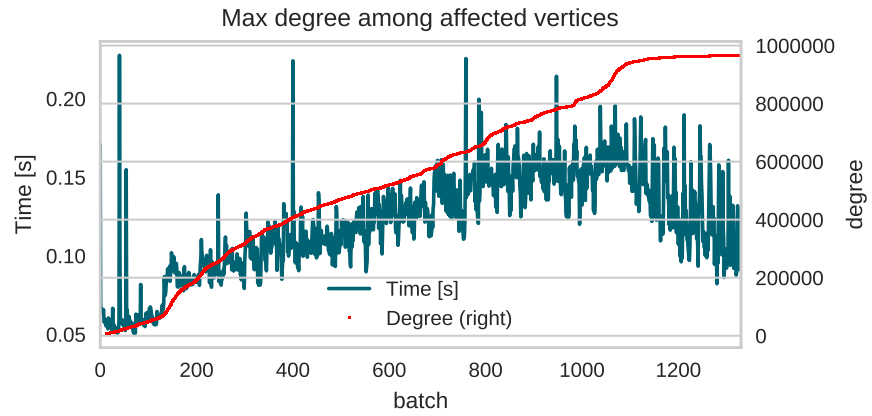
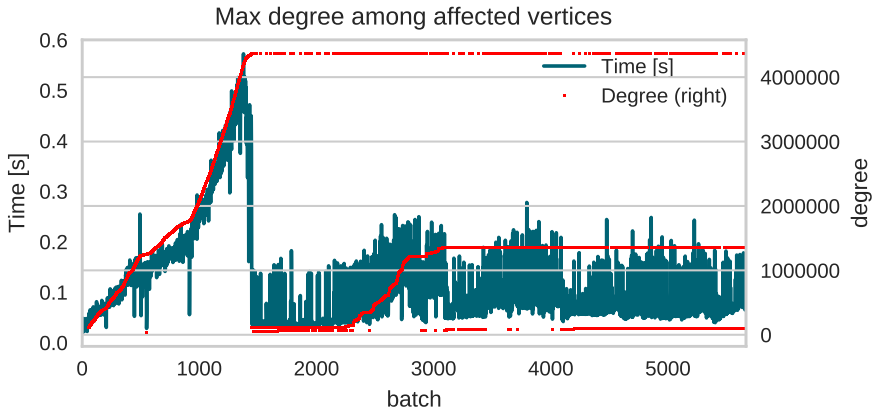


Figure 2.8: Edge deletion performance comparison between the STINGER and GAP graph engines. STINGER tolerates holes in the data structure, while GAP needs to reallocate the data structure to maintain contiguity.



(a) worldcup



(b) sc15

Figure 2.9: Adding new edges to a few highly-connected vertices has a detrimental effect on the performance of STINGER edge insertion.

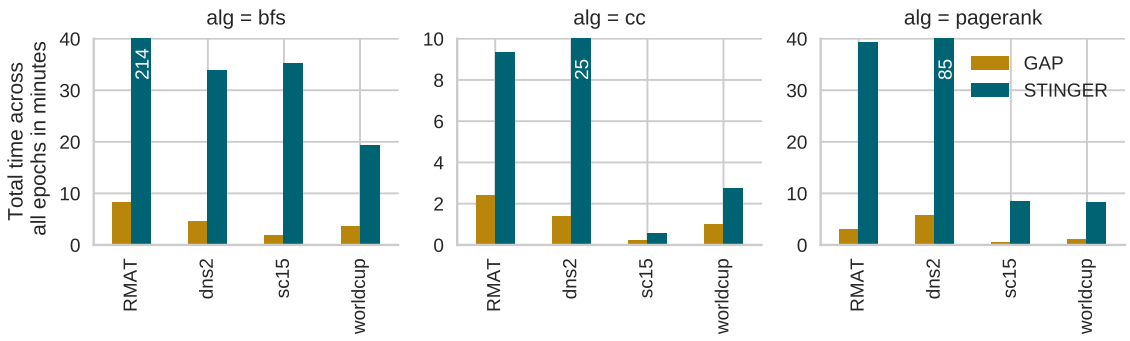


Figure 2.10: Performance of graph algorithms in GAP and STINGER. Times have been summed across all epochs. STINGER is optimized for graph updates, leading to degraded performance in algorithms.

hot-spots of real graphs. It should be possible to create a more realistic dataset by dynamically adjusting the parameters of an RMAT graph generator or randomly permuting the generated graph. This idea is further explored in Section 5.3.3.

The STINGER data structure is optimized for graph updates at the expense of graph algorithm performance. Figure 2.10 compares the run times of STINGER algorithms with their GAP counterparts, summed across all epochs. For these experiments, snapshot mode was enabled to ensure that STINGER and GAP both had a best-case memory layout. STINGER algorithms are slower than GAP in all cases. STINGER incurs additional overhead when traversing the adjacency list since it has to read edge block headers and dereference pointers to get to the next block.

Having established that STINGER is better optimized for dynamic graph processing, it will have exclusive focus for the rest of the results section. The previous results showed STINGER algorithms running on a best-case memory layout generated with snapshot mode. The following results will compare with an unsorted layout as created by incremental graph construction. Figure 2.12 shows the decline in the performance of the PageRank algorithm when using an incrementally constructed graph. Note that both sets of results use the same graph engine and the same input graphs; The only difference is the difference in memory layout introduced by incremental graph construction.

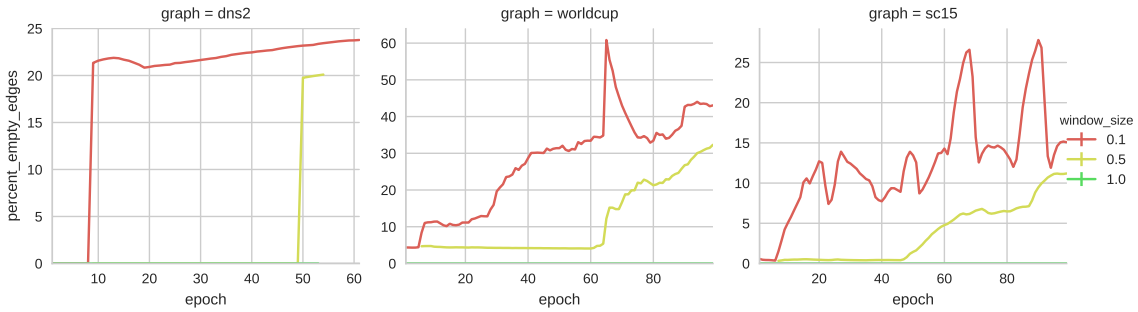


Figure 2.11: Percentage of the STINGER graph that stores deleted edges. As the size of the sliding window shrinks, more edges are deleted from the graph. Some of these edges are filled by new edges in the next batch, while others remain empty.

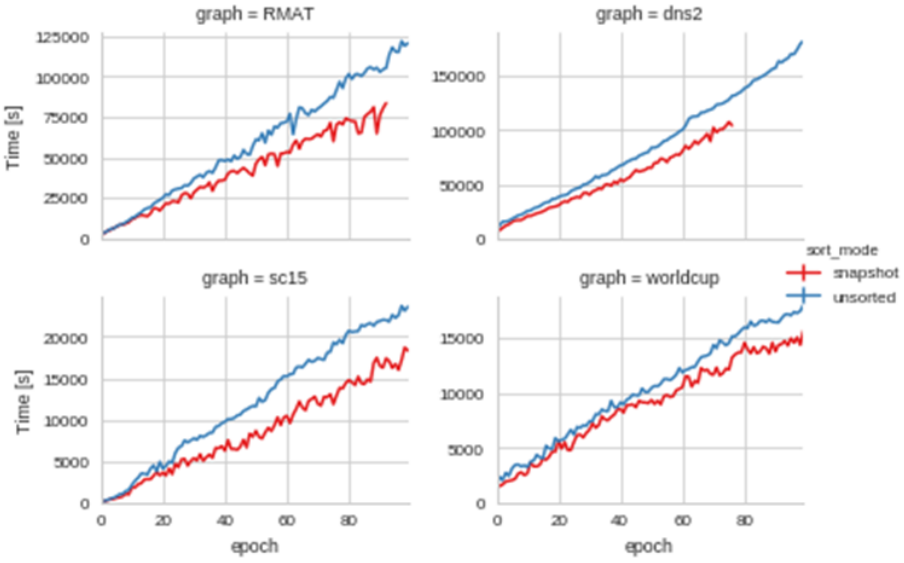


Figure 2.12: Performance of STINGER PageRank algorithm compared between unsorted and snapshot mode across window sizes of 10%, 50%, and 100%.

STINGER was further instrumented to collect diagnostic information about the level of fragmentation in the data structure after each batch of insertions and deletions. When edges are deleted in STINGER, a negative value is written to the edge location, indicating that the data should be skipped over during graph traversal. These holes still consume memory bandwidth and introduce irregularity into the iteration logic. The statistic being plotted in Figure 2.11 is the percentage of edge storage that is occupied by a deleted edge. Note that this calculation does not count edge storage that has been allocated but never written, since STINGER never looks at this data during traversal. When the window size is 100% there is zero fragmentation, since edges are never deleted. Snapshot mode also produces graphs with no fragmentation by omitting edges that would have been deleted before the current epoch. A window size of 50% produces mild fragmentation halfway through the benchmark, and a window size of 10% produces more extreme fragmentation, maxing out at 25% in the sc15 dataset and 60% in the worldcup dataset.

The STINGER algorithms are highly tolerant to an irregular data layout. Several aspects of the STINGER data structure contribute to this phenomenon. The code generated to fetch the next node in a linked list must be able to handle a worst-case random access, so it benefits little from the case where the next node is contiguous in memory. The contiguous layout used in GAP yields better performance overall. Furthermore, all of the algorithms studied in this work iterate over all of an edge's neighbors, rather than searching for a particular neighbor to explore. As a result, there is usually plenty of parallel work to cover the latency of slow memory accesses.

2.9 Conclusion

Increasing the performance of graph updates and traversals requires innovation at every layer of the HPC stack, from hardware to software. Researchers rely on benchmark scores to guide design choices and compare results with their peers. While existing graph benchmark suites have yielded many important advancements in the field of graph processing,

they are are not good proxies for streaming graph analytics in the real world. Synthetically generated graphs and graphs loaded from snapshots often fail to find system bottlenecks because they lack the irregularity of real inputs. DynoGraph expands the scope of graph benchmarks to include incremental graph construction and update, triggering a wider range of behaviors.

DynoGraph inputs are temporal graph streams that contain realistic levels of edge duplication and generate bursts of updates to high-degree vertices. The driver program incrementally loads graphs while applying deletions to generate a mature, fragmented in-memory layout that accurately models a production system in steady-state. The DynoGraph specification is flexible enough to integrate with emerging software and hardware systems, driving the future of research into streaming graph analytics.

CHAPTER 3
ACCELERATING STREAMING GRAPH ANALYTICS

This chapter evaluates several techniques for accelerating the construction and traversal of dynamic graphs. Each example explores how a different aspect of the dynamic graph workflow can be rewritten to be more efficient, or mapped onto a specialized hardware system to improve performance. Section 3.1 walks through the the design of an optimized algorithm for inserting a batch of edges into STINGER. Section 3.2 presents the design of a near-memory fetch unit for accelerating the traversal of fragmented edge lists in STINGER. The final case study in Section 3.3 introduces a new micro-benchmark for fragmented edge list traversals. It then uses multi-channel DRAM on a Knight's Landing system to simulate near-memory cores performing filtering and compression operations on the edge list before streaming data back to a host processor.

3.1 Stinger Batch Insert

3.1.1 Background

Since its initial proposal, development of the STINGER streaming graph engine has continued as an open-source project. The code has been ported to the Cray XMT [65], multi-node Intel server systems [66], and re-implemented for GPU's [67]. A recent work [68] allowed STINGER algorithms to be efficiently implemented in the Julia language.

The algorithm used to incrementally construct a STINGER graph for the results presented in section 2 was an improvement on the existing code used to run the STINGER demonstration at SC2015 [58]. The experience gained from testing the algorithms with real-world data led to insights into the worst-case performance characteristics of the existing algorithm, leading to a major improvement.

3.1.2 Parallel edge insertion benchmark in STINGER

Performing a single edge update in STINGER is $O(degree(src))$, where $degree(src)$ is the number of neighbors of the source vertex at the time of insertion. There are two steps: first, the edge list must be searched for an existing edge to update. If the edge already exists in the graph, the search may terminate early, but in the worst case, or in the case of a new edge, the entire list must be read. Because STINGER stores the neighbors of each vertex in an unsorted linked list, this process of checking for duplicates cannot be parallelized. Updating a high-degree vertex is very expensive, because a single thread must scan through the entire edge list.

Once the possibility of duplicates has been ruled out, the algorithm can insert the edge into the first available empty slot, usually at the end of the list. If there are no empty slots in the graph, a new edge block is appended to the end of the list.

The benchmark used to measure an update rate of 3 million edges per second for STINGER in [65] used the following methodology. First, the RMAT graph was read from disk and transformed into a STINGER graph. Then, a batch of 100K or 1M updates was randomly generated and inserted into the graph during a timed section. The edge update rate was computed as the total number of edge updates performed divided by the elapsed time. This methodology differs from real-world usage in several ways:

1. The base graph was created using an optimized algorithm for loading a CSR graph from disk. The blocks comprising each edge list were allocated all at once, guaranteeing that they would be contiguous in memory and that no "holes" would be present. In the SCinet scenario, the base graph was created organically as data began to be collected.

2. The benchmark assumes that a large batch of updates (hundreds of thousands to millions of edges) will be available at once. In practice, the SC application received smaller batches (thousands to tens of thousands per batch), but was still expected to

sustain an overall high rate of insertion.

3. RMAT is designed to generate a power-law graph, in which some vertices will have exponentially more neighbors than others. At each step of the algorithm, the probability of generating an edge for one of these high-degree vertices is high. Given any two high-degree vertices, it is extremely likely that an edge between the two of them will appear earlier in the stream of RMAT edges than an edge between two lower-degree vertices. This naturally sorts the edge list of each vertex by degree. As a result, when threads scan through the edge list to search for duplicates, they are much more likely to find the edge to update early in the list, avoiding the worst-case $O(degree(src))$ running time of the algorithm. In the SC dataset, the high-degree vertices don't occur until later in the stream (recall Figure 2.9b), requiring longer searches before their corresponding entries can be found.

3.1.3 STINGER batch insert algorithm

The key insight of the batch insert algorithm is in combining the work required for de-duplication between multiple updates for the the same source vertex. In the old algorithm, two threads processing an update for the same source vertex would traverse the same edge list twice in parallel. In the new algorithm, a single thread carries multiple edge updates along with it, checking each of them against each edge in the neighbor list. The new algorithm proceeds as follows:

1. The list of updates to be inserted is sorted.

2. The sorted list of updates is grouped into sub-ranges by source vertex. Sub-ranges are divided among threads according to a dynamic schedule. Large sub-ranges may be split among multiple threads; this duplicates the work of traversing the edge list in each thread, but parallelizes the work of applying the updates and inserting the edges.

3. Within a given sub-range, all updates will apply to the same source vertex. For each edge in that vertex's adjacency list, a binary search is performed on the sorted sub-range for matching updates to apply. This loop continues until the sublist of updates is empty or the end of the adjacency list is reached.

4. If there are any edges remaining, the thread returns to the beginning of the list, inserting each edge in the first available slot. When all slots are full, edges are appended to the end of the list as usual.

Python-style pseudo-code for the batch update algorithm is provided in Source Code 3.1:

Source Code 3.1: STINGER batch insert algorithm

```
def batch_insert(graph, updates):
    # Parallel sort
    updates.sort()
    # Parallel loop - each subrange may be divided further
    for src, subrange in groupby(updates, lambda edge: edge.src):
        # All edges in subrange have the same source
        for dst in graph.neighbors(src):
            # Binary search through sorted array
            if (src, dst) in subrange:
                update_edge_properties(src, dst)
                subrange.remove(src, dst)
            # Quit early if all updates have been processed
            if len(subrange) == 0:
                break
        # Insert edges that were not found
        for src, dst in subrange:
            graph.insert(src, dst)
```

3.1.4 Results

Both the original and the optimized batch insert algorithm were benchmarked using DynoGraph on the same input graphs and server system used in section 2.7. The original algorithm uses an OpenMP parallel for loop in which each loop iteration is responsible for a single edge. This algorithm was modified slightly to use a dynamic rather than a static schedule, reducing the thread imbalance.

The new algorithm dramatically improves the performance of the edge insertion when there are many new edges. Figure 3.1 plots the time taken to insert each batch. The largest improvement is seen on the sc15 dataset. Batches that used to take several minutes to insert now complete in less than a second. The only input that does not see dramatic improvement is the RMAT dataset. As discussed previously, threads rarely need to update the low-degree vertices at the end of the edge list for this graph input, so the overhead of sorting the batch before insertion ends up not being worthwhile. Figure 3.2 plots the mean insert rate across all of the `insertions` steps of the the DynoGraph benchmark for each graph input.

This case study further confirms that the real-world graphs provided by DynoGraph are better at exposing worst-case algorithm performance than synthetic datasets such as RMAT. While a simpler version of the batch algorithm was developed in [65], it was never tested with real-world inputs, and was subsequently set aside. If near-memory accelerators are evaluated with static graph benchmarks and synthetic graphs, many opportunities for optimization will be missed.

Several lessons can be learned from these experiments that are relevant to the design of near-memory accelerators for streaming graph analytics. Even though there were no dependencies between each edge insertion in the original algorithm, it was still more efficient to combine the work into a single thread. Near-memory accelerators must be able to coalesce accesses to the same memory bank. While streaming graph algorithms tend to lack locality overall, it is imperative that the architecture does take advantage of what is available. In this case, the additional step of sorting the batch was able to exploit locality

and save work later in the algorithm. While the majority of the work in a memory-centric architecture will ultimately be scattered out to the near-memory cores, the coordination and merging of similar pieces of work will be vital to maximizing performance.

3.2 Edge Block Prefetcher

3.2.1 Motivation

There can be significant latency between the memory controller and the host processor, especially in systems where the memory controller is not located on the CPU die. This latency is usually hidden when data is transferred in large bursts and stored in the on-chip cache, but the traversal of linked data structures in the DynoGraph workloads aggravates this problem. Unless the algorithm calls for a large amount of computation per edge, the processor will spend most of its time stalled while waiting to dereference the pointer to the next block. Hardware prefetchers [69, 70] that predict the next access based on history are unable to find a pattern in this stream of pointer accesses, and software techniques [71, 72] rely on finding enough work to overlap with the fetch latency. In contrast, a content-directed prefetcher examines fetched data for pointers to prefetch. One successful approach [73] searches the cache for data that are likely to be pointers, and issues prefetches to these locations.

The Edge Block Prefetcher (EBP) is a content-directed prefetcher that accelerates edge list traversals in STINGER. It is co-located with the last level cache (LLC). The EBP is triggered by software, after which it operates automatically to bring data into the LLC before it is requested by the processor. The EBP uses knowledge about the structure of the data being traversed to improve accuracy, while shortening the round-trip time for each pointer dereference.

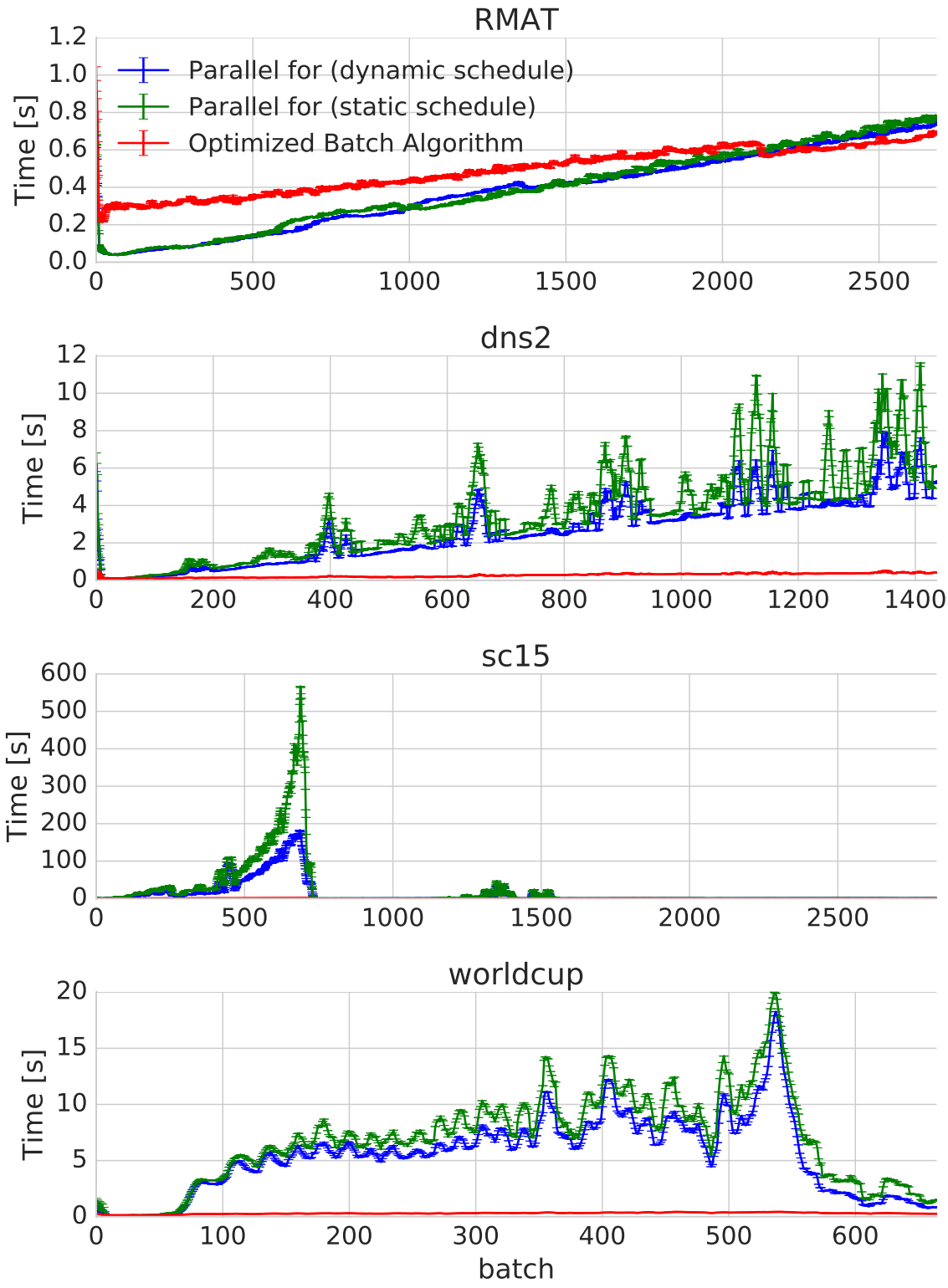


Figure 3.1: Performance of improved STINGER batch insert algorithm across batches of DynoGraph inputs

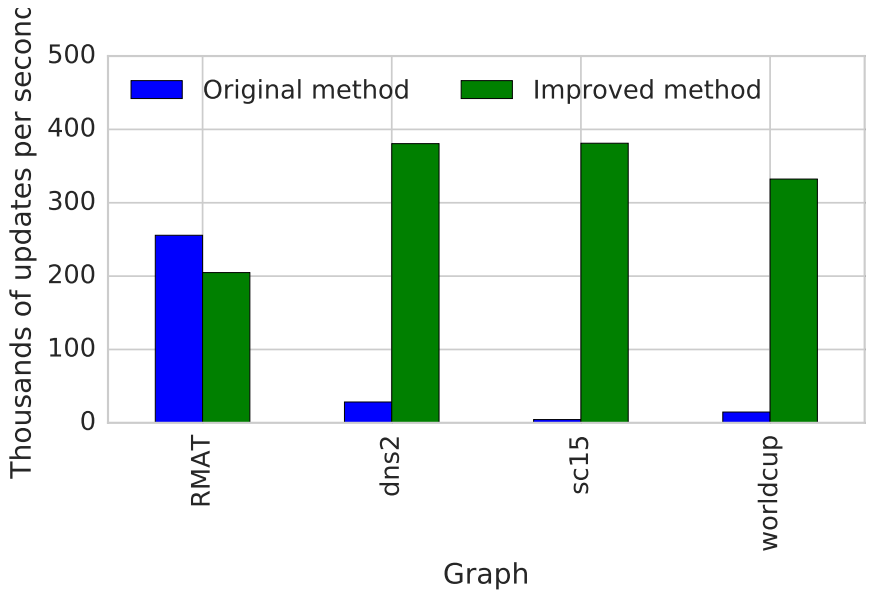


Figure 3.2: Performance of improved STINGER batch insert algorithm relative to previous implementation

3.2.2 Related Work

Impulse [74] implements application-controlled memory address space transformation at the memory controller, allowing applications to more efficiently fill cache blocks when data has non-unit stride. The Indirect Memory Prefetcher [75] detects and prefetches indirect accesses of the form `A[B[i]]`, intelligently prefetching the index array `B` in order to generate second-level prefetches for the outer array `A`. In "Meet the Walkers" [76], the authors propose a set of simple RISC cores at the MMU that perform hash table lookups for database applications. Refer to [77, 78] for more examples of near-memory fetch units.

3.2.3 Software interface

The software interface of the EBP unit consists of a single new instruction that is inserted before the beginning of a linked list traversal. It accepts four arguments that specify the structure of the linked list: `base`, `start`, `size`, and `depth`. Fundamentally, a linked list stores a pointer to the next node in the list, which can be either a raw pointer or an index into a pool of pre-allocated nodes. This interface is designed to handle both cases.

```
// Example A
class NodeA {
  NodeA *next;
  uint64_t data[63];
};

traverseList(NodeA *head) {
  EBP(0, head, 1, sizeof(NodeA));
  for (NodeA *p = head; p; p=p->next) {
     doWork(p->data);
  }
}

// Example B
class NodeB {
  NodeB *next;
  uint64_t foo;
  uint64_t data[126];
};

traverseList(NodeB *head) {
  EBP(0, head, 1,
      sizeof(NodeB*) + sizeof(uint64_t));
  for (NodeB *p = head; p; p=p->next) {
     doWork(p->foo);
  }
}
```

```
// Example C
class NodeC {
  uint64_t data[7];
  NodeC *next;
};

traverseList(NodeC *head) {
  EBP(offsetof(class NodeC, next), head, 1, sizeof(NodeC));
  for (NodeC *p = head; p; p=p->next){
     doWork(p->data);
  }
}

// Example D
class NodeD {
  uint64_t next;
  uint64_t data[63];
};

traverseList(int head, NodeD *nodePool){
  EBP(&nodePool[0], head,
      sizeof(NodeD), sizeof(NodeD));
  for (NodeD *p = &nodePool[head]; p; p=&nodePool[p->next])
  {
     doWork(p->data);
  }
}
```

Figure 3.3: Examples of using the Edge Block Prefetcher software interface

Figure 3.3 gives examples of how this API can handle four different singly-linked data structures. Note that these examples assume a 64-bit machine with 64-byte cache blocks.

Example A is a simple linked list traversal. The EBP call specifies a base of zero since the next pointer is at the beginning of the node, and a size of 1 since raw pointers are in use. The traversal starts with the head of the list, and fetches the entire node. In Example B only a portion of the linked list is prefetched. Example C shows how to handle situations where the next pointer is not at the beginning of the structure. The `offsetof` macro returns the byte offset of a field within a struct. The nodes in Example D each store an index into a node pool instead of a raw pointer. The base is set to the beginning of the node pool, while the start parameter is now the index of the first node. The size parameter is set to the size of a node so that the EBP can do table indexing.

3.2.4 Functional Description

When the processor executes the new `edgeBlockPrefetch` instruction, it generates a tagged load which is intercepted by the LLC. This triggers the EBP unit, which stores the request parameters and issues a request to the LLC for the cache block that contains the

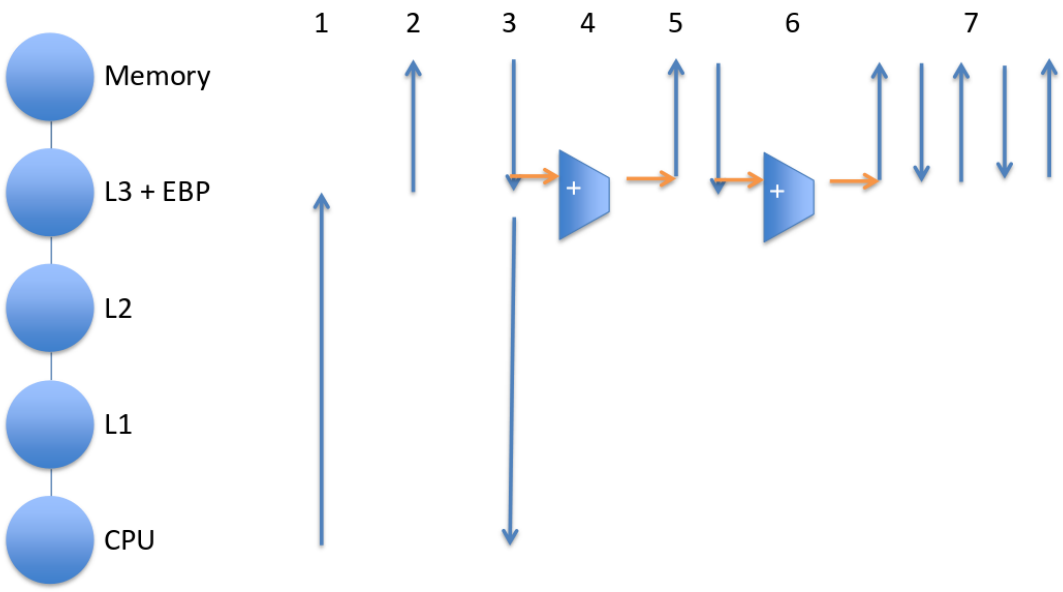


Figure 3.4: Operation of the Edge Block Prefetcher

first element of the list. When the block arrives (or is found in the cache), the EBP extracts an index from the block and computes the next pointer using the indexing parameters from the initial request. Lastly, the EBP performs a virtual-to-physical translation on the pointer and sends it to the LLC as a prefetch request. The EBP also sends prefetch requests for blocks directly following the pointer, as specified by the depth parameter; however only the first block is recorded in the EBP request table. This process of walking the linked list pointers continues until the extracted pointer is zero, indicating the end of the list. Multiple lists can be fetched simultaneously by a single EBP unit.

If the EBP is able to fetch blocks faster than the processor requests them, eventually the cache will be full of prefetched blocks that have not been used. The normal LRU behavior of the LLC would evict blocks before they have a chance to be used. Instead, the EBP

pauses the prefetching and allows the processor to catch up. No new prefetches will be issued until a miss is seen for the block that caused the pause.

3.2.5 Auto-depth prefetch

A programmer can use the depth parameter to tell the EBP unit to fetch only the first few fields in a struct node, as in Example B from Figure 3.3. The EBP was further extended to perform auto-depth prefetching for structs that have variable-length fields within the node. While STINGER edge blocks have a fixed size, newly allocated blocks may be mostly empty. The number of edges actually present in an edge block is stored in the header. When inspecting a node for the next pointer, the EBP also extracts this field and uses it to trigger fetches of valid data in the edge block. Using auto-depth prefetch also slightly increases the latency of fetching the first block. Instead of requesting the entire node along with the header, the EBP must first fetch the header, inspect it for the depth, and then fetch the rest of the node along with the header for the next node. The advantage of auto-depth prefetch is that it conserves space in the cache and reduces bandwidth across the memory bus.

3.2.6 Experimental Methodology

The EBP was implemented as described in the gem5 [79] full-system simulator. Gem5's Ruby memory model was chosen for its flexibility and detail of modeling coherence protocols in a domain-specific language. The existing MESI Three Level coherence protocol was modified to add the new states and transitions necessary to interact with the EBP. The EBP was tested with an older version of the STINGER DynoGraph benchmark introduced in chapter 2, using smaller graph inputs drawn from Twitter data and citation networks.

Edge block prefetch instructions were inserted into the graph traversal code. While the base, start, and size arguments were fixed to accommodate the STINGER data structure. several values for the depth parameter were tested. Setting the depth to 512 bytes asks

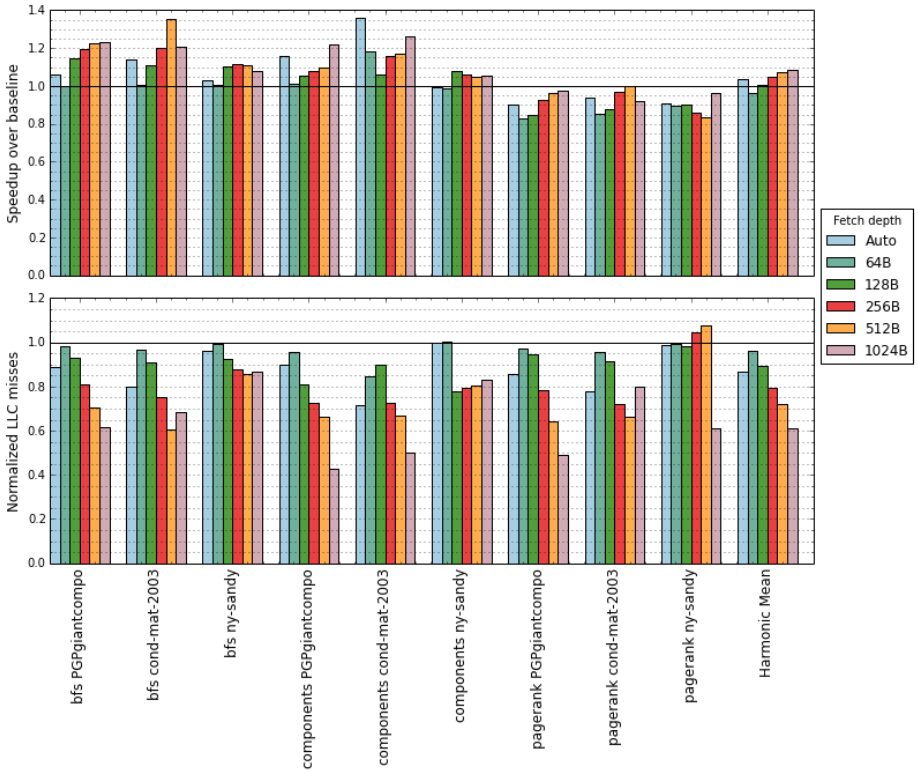


Figure 3.5: Performance of the Edge Block Prefetcher

the EBP to fetch the entire edge block, predicting that the block will be full. Setting the depth to 64 bytes asks the EBP to only fetch the first two edges in the block, predicting that blocks will be mostly empty. Other intermediate values of 128 bytes and 256 bytes were also tested. Auto-depth prefetch checks the size first, then fetches only the edges that are actually present.

A prefetch depth of 1024 bytes was also tested. This effectively fetches an entire edge block along with the edge block immediately after in the edge block pool. Edge blocks for the same vertex tend to be claimed sequentially, however there is no guarantee that the next edge block is connected to the first, nor that it will ever be needed by the graph traversal algorithm. In particular, edge blocks are not likely to be sequential after the graph has undergone many insertions and deletions.

3.2.7 Results

Overall the EBP is able to greatly improve the overall runtime of the program. Some configurations see a 35% speedup over the baseline. The 1024-byte depth fetch performed best overall, with an average speedup of almost 10%. The cond-mat-2003 input saw the greatest improvement. The only workload that did not experience an overall speedup is PageRank. This is surprising since the overall number LLC misses decreased. There are a number of possible reasons for the slowdown. Compared with the other algorithms, PageRank stores a larger number of auxiliary tracking structures in addition to the graph itself. EBP does not attempt to preserve these structures in the cache, so they may be evicted when EBP runs ahead. Thus the overall miss rate may decrease even though the algorithm is still regularly stalled waiting for critical data.

EBP is able to reduce the overall runtime of the program because it converts long-latency DRAM reads into last level cache hits. Fig. 3.5 shows the overall reduction in LLC misses for each benchmark. The reduction in LLC misses is directly correlated with performance gains. Increasing the prefetch depth generally reduces the number of LLC misses, but there is often an inflection point after which the number of misses starts to rise again. For example, bfs cond-mat-2003 has an ideal depth of 512 bytes, while components ny-sandy has an ideal depth of 128 bytes. In particular, ny-sandy seems especially sensitive to over-fetching, which brings useless blocks into the cache and evicts other blocks that may be useful.

The pie charts in figure 3.6 break down the effectiveness of the EBP in each experiment. The total pie area represents all the prefetches that the EBP issued for each run. Note that the total number of prefetches issued for each configuration are not the same. There are several possible outcomes for a prefetch request, which are color-coded in the legend. If the prefetched request is later accessed by a demand request, it is counted as a hit (green). If the demand request arrives while the prefetch is still in-flight, it is counted as a partial hit (yellow). Partial hits are still useful since they reduce the latency of a DRAM access.

If the prefetch request hits in the LLC, it is counted as a cache hit (blue). A large fraction of prefetch cache hits indicates that the graph traversal was already getting a lot of cache hits even without the addition of EBP. Any prefetches that do not fall into the categories mentioned above are marked as useless (white). Components sees the highest number of cache hits. Auto-depth prefetch sees the highest percentage of prefetch hits in each configuration.

3.3 Empirical performance of near-memory accelerator

3.3.1 Motivation

The paradigm of near-data processing can be summarized as "slow cores near fast memory, with a slow link to fast cores". For example, in order to aggregate multiple Hybrid Memory Cubes into a single system, as in the SB-850 board available from Micron [80], several cubes may be daisy-chained together as depicted in figure 3.7. In this situation, the link between the host processor and the first cube in the chain limits the peak memory bandwidth to that of a single cube. If lightweight processors are embedded in the logic layer of the HMC, as proposed in [25, 26, 81], these cores will have plentiful bandwidth but limited computational power.

In view of these unique design constraints, a custom near-memory accelerator is designed to improve the performance of dynamic data structure traversals. The goal is to offload the irregular traversal of the edge block list to the near-memory cores, which will generate many memory requests in parallel to hide latency and saturate the bandwidth of the local memory channels. As the graph data is fetched, a stream of data will be continuously sent back to the host processors. This stream will omit the internal pointers and gaps in the graph data structure, effectively creating a compressed format more similar to CSR. Host cores that process this dense, regularized stream of data will experience fewer branch mispredictions and pipeline stalls, leading to increased ILP and more efficient use of the interconnect bandwidth.

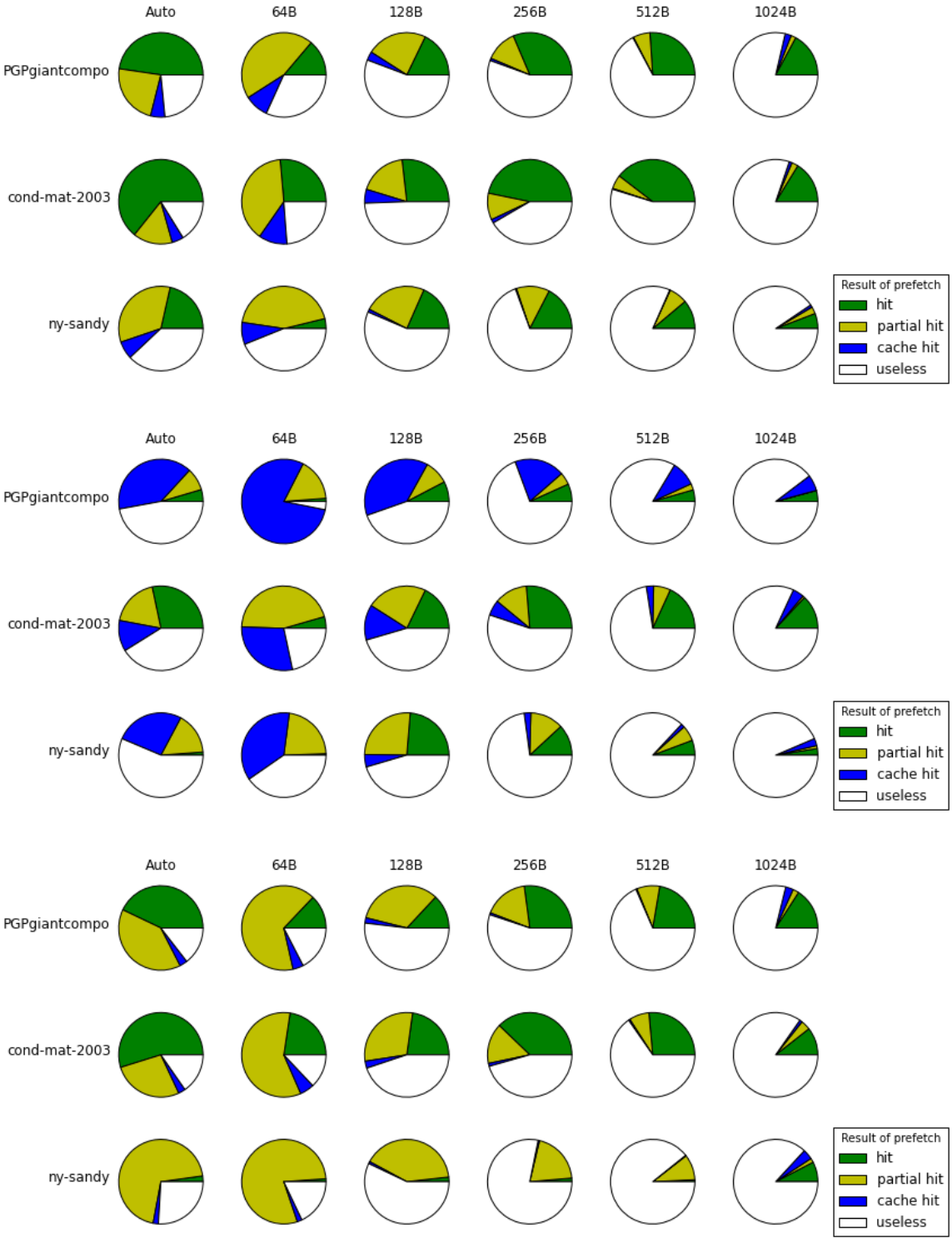


Figure 3.6: Distribution of outcomes for each prefetch issued.

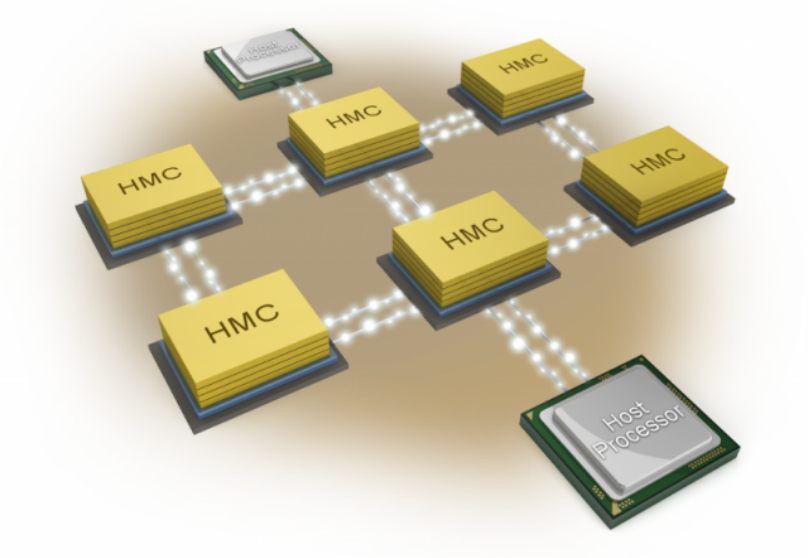


Figure 3.7: A daisy chain configuration of multiple Hybrid Memory Cube devices. Reproduced from www.extremetech.com

3.3.2 Related Work

The Hybrid Memory Cube (HMC) is a 3D-stacked memory technology developed by Micron [22]. DRAM dies are stacked vertically on top of a logic die. The stack is joined together with through-silicon vias (TSV's). The cube is divided into 32 vertical columns called vaults; each vault forms an independent memory channel with 10GB/s of bandwidth. The logic die contains a memory controller for each vault, along with inter-vault routing and four off-chip serial links. These links can be connected directly to a processor core via a silicon interposer, or daisy-chained together to form a network of cubes.

The idea of placing logic in the memory array was first proposed by Harold Stone in 1970 [82]. Yet the fact that silicon optimized for memory density is not well-suited for efficient logic circuits has limited the effectiveness of PIM development in the past. While truly "in-memory" approaches are still viable [83, 84], the logic layer of the HMC is a much more attractive location for positioning near-memory cores, opening the door for more research in this area. The HMC ships with simple atomic operations implemented in the logic layer, which can be integrated directly into existing graph workloads [28].

Researchers are already beginning to look forward to fully customizing the processing capabilities of the HMC logic layer. While it is clear that real-time analytics applications will benefit greatly from these designs [85], there are still many important design choices to be made.

Tesseract [25] connects 16 PIM-enabled HMC's into a mesh to form a parallel graph processing architecture with a total internal memory bandwidth of 8 TB/s. Each vault is equipped with an in-order core, a small L1 cache, a stream prefetcher, and a special content-directed prefetcher. Accessing data in remote vaults is done through remote function calls. Gao et al. [26] also describe a PIM framework for accelerating analytics workloads within HMC. Unlike Tesseract, this work allows the in-memory cores to access any vault through a lightweight, software-assisted coherence layer, additionally coordinating host and PIM cores through virtual memory. Other graph-specific accelerators include Graphicionado [86].

The Active Memory Cube [87] proposal envisions the in-memory cores having SIMD vector processors, suitable for scientific workloads such as DGEMM and DAXPY. Kersey et al. [88] prototyped an HMC system with GPU-like soft cores running in an FPGA. Guo et al. [89] proposed a library for accelerating common Intel Math Kernel Library (MKL) operations on the HMC.

Gokhale et al. [81] propose a different kind of near-memory accelerator. Instead of delegating work to the logic-layer threads, the threads present a coalesced or reshaped view of memory that more closely matches the access pattern of the application. Host processors can request the creation of such a "view buffer" through a series of API calls. The logic-layer threads efficiently assemble fragmented data words into a contiguous buffer using the intra-HMC bandwidth, then optionally write each byte back to its original location. In a similar fashion, the SPARC M7 processor [90] provides eight hardware accelerators for decompressing and filtering data for in-memory database traversals.

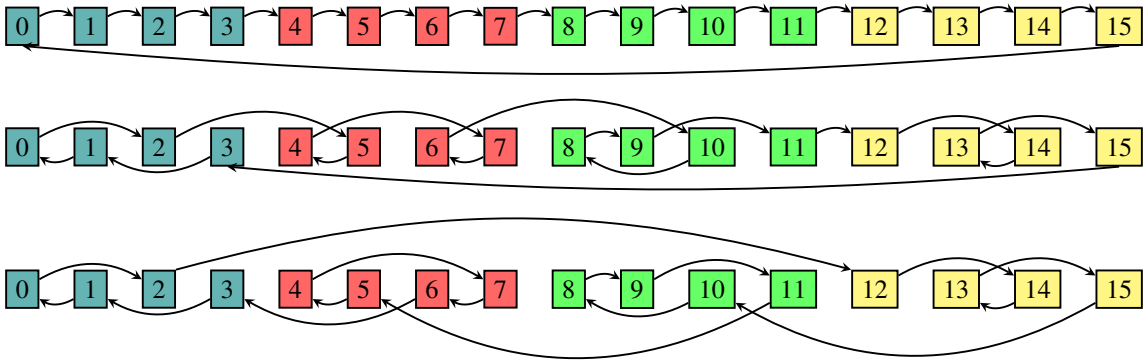


Figure 3.8: (top) An ordered linked list, in which consecutive elements have sequential memory addresses, (middle) A linked list with a intra-block shuffle permutation applied to randomize the ordering of elements within a block. Note that all elements within a block are accessed before jumping to the next block. (bottom) A linked list with a full block shuffle permutation applied. Not only are the elements within a block shuffled, but the traversal order of the blocks themselves has also been randomized.

3.3.3 Pointer Chasing Benchmark

A new benchmark, called "pointer chasing", was designed to simulate the traversal of an edge list data structure. In this benchmark, each thread sums up all the elements in a linked list. Each element consists of an 8-byte payload and an 8-byte pointer to the next element. After the elements of this linked list are grouped into blocks, their ordering is randomized. This permutation may be applied to the ordering of the elements within each block (`intra_block_shuffle`), or the ordering of the blocks themselves (`block_shuffle`), or both (`full_block_shuffle`). Figure 3.8 explains the list initialization further.

The block size in this benchmark can be varied to simulate different levels of spatial locality that may arise in a workload. A small block size creates clusters of contiguous elements that will fit in a cache line or a single DRAM row buffer, similar to a STINGER edge block. A larger block size creates a partitioned linked list, creating a large number of hops within a single memory bank or partition of a PGAS-style system before crossing boundaries. At the extremes (block size of 1 or block size equal to the number of elements),

the pointer chasing benchmark simulates a worst-case memory fragmentation scenario that can arise when small list elements are dynamically allocated and deallocated from a single shared memory pool.

The pointer chasing benchmark was designed to have three key properties.

- Data-dependent loads: Memory-level parallelism is severely limited since each thread must wait for one pointer dereference to complete before accessing the next pointer
- Fine-grained accesses: Spatial locality is restricted since all accesses are at a 16B granularity. This is smaller than a 64B cache line on x86 platforms, and much smaller than a typical DRAM page size (around 8KB).
- Random access pattern: Since each block of memory is read exactly once in random order, caching and prefetching are mostly ineffective.

The pointer chase benchmark is quite similar to the RandomAccess benchmark [91], also known as GUPS. However pointer chasing does not allow lookahead, as the access order is embedded in the list itself rather than being calculated from a random stream. Furthermore pointer chasing does not modify the list elements.

3.3.4 Experimental Setup

The accelerator design can be tested on a larger scale at greater speed using native execution on physical hardware. Intel's recently released Knights Landing (KNL) [92] processor consists of 72 cores arranged in a grid on a single die, interconnected with several memory controllers by means of a mesh network. In addition to external DDR4 DRAM, KNL also includes 16 GB of on-die multi-channel DRAM (MCDRAM), which delivers over 450 GB/s of STREAM [4] bandwidth. The KNL cores enjoy a direct, high-bandwidth connection to the MCDRAM, just like the logic-layer cores in the proposed HMC variant. If a dynamic graph data structure is stored in MCDRAM (as a proxy for the vaults across several Hybrid Memory Cubes) and a compressed stream is written to DDR memory (as a

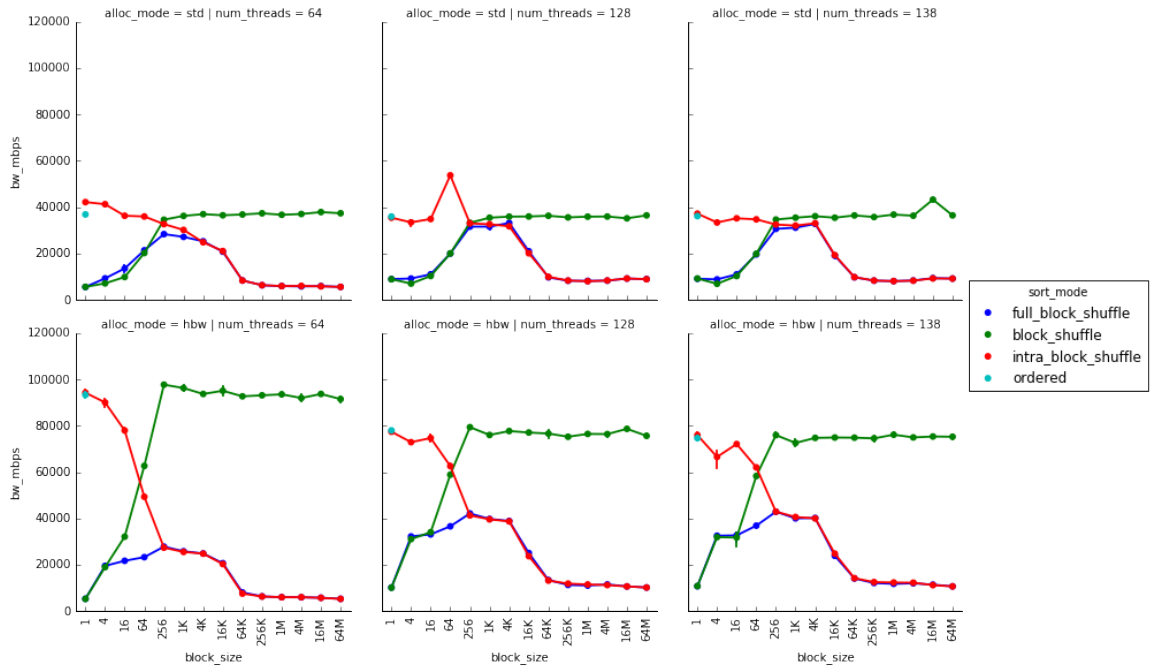


Figure 3.9: Performance of pointer chasing benchmark on a KNL system. Top row uses the external DDR4 memory, while the bottom row uses the on-die MCDRAM.

proxy for the link to the host cores), then the accelerator application can be implemented in software on the KNL system as a proof of concept. The processing power of the near-memory cores can be varied by arbitrarily limiting the thread count and the maximum level of per-core simultaneous multi-threading.

3.3.5 Results

In order for the near-memory accelerator to be effective, it must be able to traverse the linked-list in MCDRAM faster than the host core can traverse the list in DDR4 RAM. While the MCDRAM provides STREAM bandwidth that is 4x higher than DDR4, it provides no performance improvement on the `full_block_shuffle` variant of the pointer chasing benchmark. If elements within a block are in order (`block_shuffle`, the MCDRAM accelerator can achieve peak bandwidth with a block size of 256, or 4KB of contiguous accesses between each random jump. But when elements within a block are not in order, bandwidth utilization falls to below 40% of peak, even for the best block size configuration.

3.4 Conclusion

The results in section 3.3.5 highlight a key point about emerging memory systems: more bandwidth is not always helpful for irregular applications. The MCDRAM relies on sequential accesses within a 4KB page in order to deliver peak bandwidth. The system was unable to benefit from the additional MLP provided by dozens of threads generating independent memory requests.

An ideal processing-near-memory solution for streaming graphs and other irregular applications needs to provide more than just additional STREAM bandwidth. The near-memory cores should be able to maintain and switch between a large number of application-aware thread contexts in order to hide latency and maximize the utilization of the local memory channels. Such a system must be able to sustain a high level of bandwidth when receiving a large number of fine-grained requests, while taking advantage of spatial locality where it exists. Finally, the system must be able to deal with traversals that involve multiple near-memory cores when a data structure stretches across memory banks.

CHAPTER 4
CHARACTERIZATION OF THE EMU CHICK

4.1 The Emu Architecture

The Emu architecture focuses on improved random-access bandwidth scalability by migrating lightweight, *Gossamer* threads to data and emphasizing fine-grained memory access. A general Emu system consists of the following processing elements, as illustrated in Figure 4.1:

- A common *stationary* processor runs the operating system (*e.g.* Linux) and manages storage and network devices.
- *Nodelets* combine narrowly banked memory with several highly multi-threaded, cacheless *Gossamer* cores to provide a memory-centric environment for migrating threads.

These elements are combined into nodes that are connected by a RapidIO fabric. The current generation of Emu systems include one stationary processor for each of the eight nodelets contained within a node. System-level storage is provided by SSDs. More specific details about some of the prototype limitations of the Emu Chick prototype are discussed in Section 5.3. A more detailed description of the Emu architecture is available elsewhere [32].

For programmers, the Gossamer cores are transparent accelerators. The compiler infrastructure compiles the parallelized code for the Gossamer ISA, and the runtime infrastructure launches threads on the nodelets. Currently, one programs the Emu platform using Cilk [93]. The current compiler supports the expression of task or fork-join parallelism through Cilk's `cilk_spawn` and `cilk_sync` constructs, with a future Cilk Plus software release in progress that would include `cilk_for` (the nearly direct analogue of OpenMP's

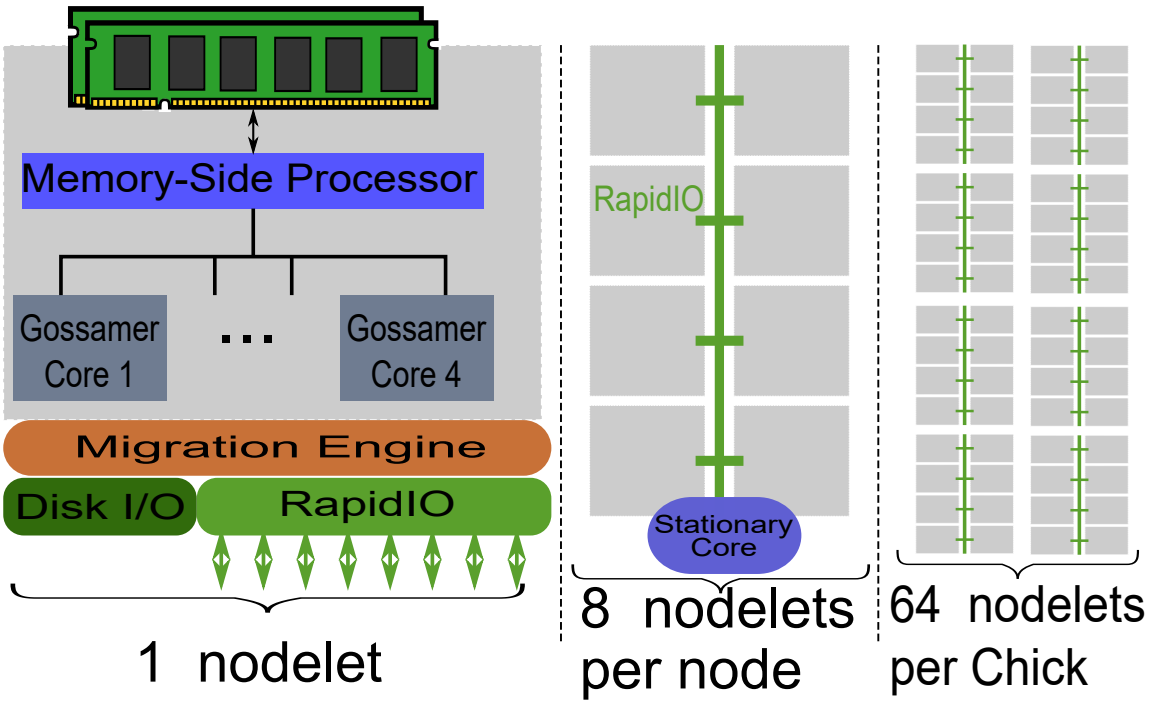


Figure 4.1: Emu architecture: The system consists of *stationary* processors for running the operating system and up to four *Gossamer* processors per nodelet tightly coupled to memory. The cache-less Gossamer processing cores are multi-threaded to both source sufficient memory references and also provide sufficient work with many outstanding references. The coupled memory's narrow interface ensures high utilization for accesses smaller than typical cache lines.

`parallel for`). Many existing C and C++ OpenMP codes can translate almost directly to Cilk Plus.

A launched Gossamer thread only performs local reads. Any remote read triggers a migration, which will transfer the context of the reading thread to a processor local to the memory channel containing the data. Experience on high-latency thread migration systems like Charm++ identifies migration overhead as a critical factor even in highly regular scientific codes [94]. The Emu system keeps thread migration overhead to a minimum by limiting the size of a thread context, implementing the transfer efficiently in hardware, and integrating migration throughout the architecture. In particular, a Gossamer thread consists of 16 general-purpose registers, a program counter, a stack counter, and status information, for a total size of less than 200 bytes. The compiled executable is replicated across the cores to ensure that instruction access always is local. Limiting thread context size also reduces the cost of spawning new threads for dynamic data analysis workloads. Any operating system requests are forwarded to the stationary control processors through the service queue.

The highly multi-threaded Gossamer cores, which are reading only local memory, do not need caches nor, therefore, cache coherency traffic. Additionally, "memory-side processors" provide atomic read or write operations that can be used to access small amounts of data without triggering unnecessary thread migrations. A node's memory size is relatively large (64 GiB) but with multiple, narrow memory channels (8 channels with 8-bit interfaces), in order to extract weak spatial locality from data analysis kernels while maintaining low-latency read and write operations. The high degree of multi-threading also helps to cover the migration latency of the many threads. The Emu architecture is designed from the ground up to support high bandwidth utilization and efficiency for demanding data analysis workloads.

4.1.1 Related Work

The Cray XMT [30] architecture was designed for superior performance on random access benchmarks like GUPS. Each processor runs 128 threads and can maintain a total of 1024 outstanding memory requests at once. Caching of non-local data is not implemented, simplifying the network design and eliminating the need to implement cache coherence. STINGER was originally optimized for the Cray XMT [95].

The GoblinCore-64 (GC64) [31] architecture is designed for memory-intensive applications that access memory with non-unit stride. Fine-grained task spawning is built into the ISA, allowing effective latency hiding when many tasks access the HMC concurrently. In the latest proposal, additional threads perform dynamic memory request coalescing in order to maximize the size of HMC requests and minimize redundant fetching. Other works on supporting a large number of lightweight threads include Qthreads [96] and the Swarm architecture [97].

4.1.2 Emu Chick Prototype

The Emu Chick prototype is still in active development. The current hardware iteration uses an Arria 10 FPGA on each node card to implement the Gossamer cores, the migration engine, and the stationary cores. Several aspects of the system are scaled down in the prototype Emu system versus the next-generation Emu system which will use larger and faster FPGAs to implement computation and thread migration. The Emu Chick prototype currently has the following features and limitations:

- The prototype system has one Gossamer Core (GC) per nodelet with a concurrent max of 64 threads. The next-generation system will have four GC's per nodelet, supporting 256 threads per nodelet.
- The prototype GC's are clocked at 150MHz rather than the planned 300MHz in the next-generation Emu system.

Table 4.1: Specifications for current and future Emu systems

	Emu Nodelet		Emu Node Card (8 nodelets)		Emu Chick (8 nodes)		Emu1 Rack (256 nodes)
	Current	Future	Current	Future	Current	Future	
# of cores	1	4	8	32	64	256	8192
# of threads	64	256	512	2048	4096	16384	>2 million
Memory Capacity	2 GiB	8 GiB	16 GiB	64 GiB	128 GiB	512 GiB	16 TiB
# of channels	1	1	8	8	64	64	2048
Memory bandwidth	140 MB/s	2.5 GB/s	1.2 GB/s	20 GB/s	8 GB/s	160 GB/s	5.12 TB/s

- The DDR4 DRAM modules are clocked at 1600MHz rather than the full 2133MHz allowed by the specification.

- Firmware bugs in the inter-node routing engine limit us to using one node (8 nodelets, single-node) at a time, rather than the full 8 nodes (64 nodelets, multi-node) in the Emu Chick.

- The current Emu software version provides support for C++ but does not yet include functionality to translate Cilk Plus features like `cilk_for` or Cilk `reducers` [98] to Emu threads. For this reason, all benchmarks are currently implemented using `cilk_spawn`. However, the use of `cilk_spawn` does allow for more control over spawning strategies in Section 4.2.

- Each node card in the system uses approximately 40 Watts, and the entire system uses approximately 350 Watts.

- Both the hardware and the simulator allow for unlimited thread spawns, but the simulator also includes an unlimited size buffer to store inactive threads. In practice, this means that recursively spawning threads on the hardware can lead to node crashes. Thread spawning limitations are discussed more in Section 5.5.

4.2 Primitives

4.2.1 Exploiting parallelism within an Emu nodelet

As mentioned previously, the Emu software distribution does not provide a working implementation of `cilk_for`. While most of the irregular applications discussed in this work focus on traversing linked data structures, it is still desirable to use loop-based parallelism to quickly initialize arrays and perform element-wise operations. Consider the ADD kernel of the STREAM benchmark, which computes the vector sum of two arrays. Source Code 4.1 shows a simple serial implementation. If `cilk_for` were available, it could easily parallelize this loop as in Source Code 4.2. The grain size argument specifies a minimum number of elements to give to each thread.

Source Code 4.1: Serial for loop

```
// Initialized with malloc(sizeof(long) * n)
long *a, *b, *c;
for (long i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
}
```

Source Code 4.2: Cilk parallel for loop

```
// Initialized with malloc(sizeof(long) * n)
long *a, *b, *c;
#pragma cilk grainsize=grain
cilk_for (long i = 0; i < n; ++i) {
    c[i] = a[i] + b[i];
}
```

In order to parallelize the first loop by hand, a secondary loop is added to split up the iteration space and spawn a thread for each sub-range, as in Source Code 4.3. The vector add logic is moved to a worker function so that multiple copies of it can be spawned. This

loop achieves good parallel speedup when all of the array elements are local to a single nodelet. A natural optimization would be to spawn the threads recursively rather than one at a time, as the compiler would do in the case of `cilk_for`. The code for this is not shown because it didn't provide any speedup over serial thread spawn (see Figure 4.6), and it would introduce an additional auxiliary function that would make the listings longer and harder to understand.

Source Code 4.3: Implementation of parallel for loop without cilk_for

```
// Initialized with malloc(sizeof(long) * n)
long *a, *b, *c;
for (long i = 0; i < n; i += grain) {
    long begin = i;
    long end = begin + grain <= n ? begin + grain : n;
    cilk_spawn worker(begin, end, a, b, c);
}
cilk_sync;
void worker(long begin, long end, long* a, long* b, long* c) {
    for (long i = begin; i < end; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

4.2.2 Distributed memory layouts and spawn trees

Parallelizing this loop across all the nodelets in the system requires changing the data layout. All of the elements in an array allocated with `malloc` are on a single nodelet, and must be read by threads running on local GC's. The Emu software libraries provide several functions for distributed memory allocation, which are described in figure 4.2. In a striped array, consecutive elements are on different nodelets. `mw_malloc1dlong` allocates a

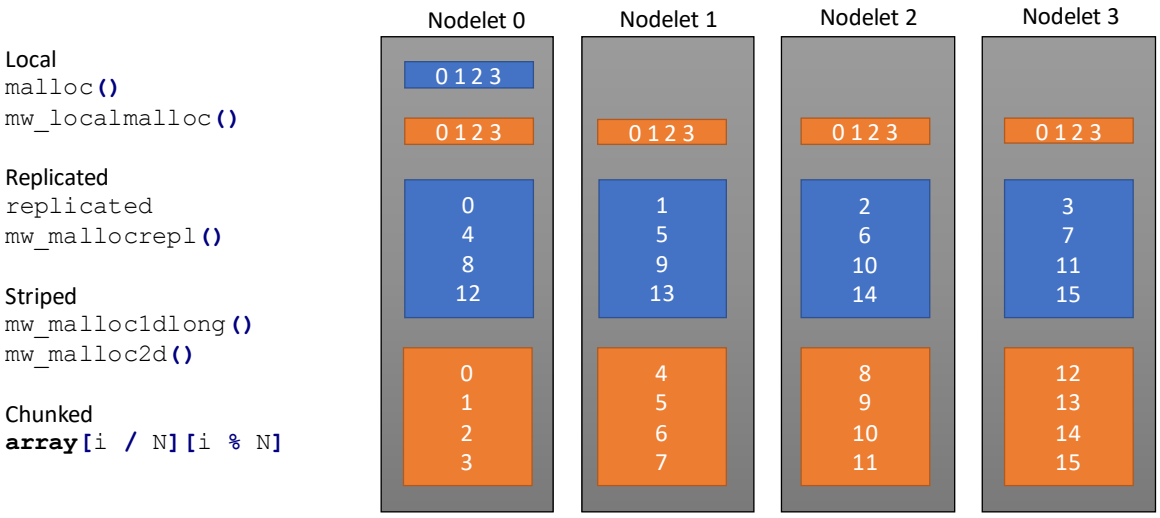


Figure 4.2: Distributed memory allocation on the Emu Chick

striped array of 8-byte integers which can be conveniently accessed using the same syntax as a regular array. If the three local arrays in the STREAM benchmark are replaced with striped arrays, the code in all of the previous listings will compile, run, and produce the correct results. However the performance will be poor for several reasons.

First, the local loops create all of the worker threads on a single nodelet. While a recursive spawn shows little benefit on a single nodelet, a recursive spawn tree that involves multiple nodelets, as shown in figure 4.3, is essential to quickly spinning up enough threads to achieve peak performance. The next listing uses a two-level spawn strategy: first a thread is spawned out to each nodelet, which in turn performs a serial spawn on each nodelet. When the Emu compiler detects a pointer to a remote nodelet in the argument list, it performs a remote spawn rather than a local spawn.

Second, the stride of the loops above do not match the allocation of the array. Each iteration of the loop will force a migration to the next nodelet. To correct this, Source Code 4.4 staggers the starting point of each thread according to its nodelet id, and increments the loop counter by the number of nodelets. This is somewhat reminiscent of the way GPU code is written to coalesce memory accesses from each thread in a warp.

A final change that must be made to enable good performance is to replicate the pointers

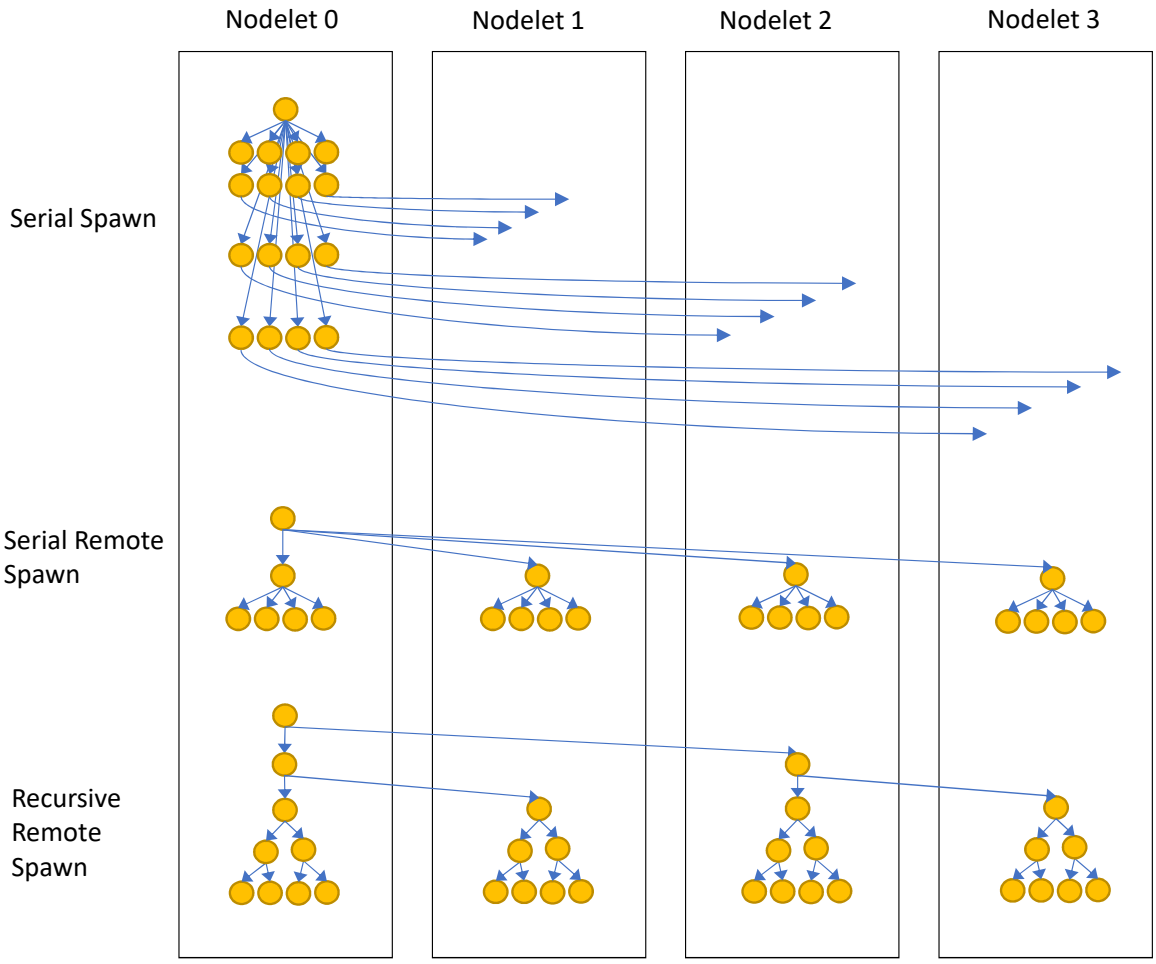


Figure 4.3: Distributed spawn trees on Emu. For brevity, this example assumes 16 threads will be created across 4 nodelets. Not shown is the local recursive spawn, which creates 16 threads locally using a recursive spawn tree before allowing them to migrate away as in the serial spawn.

to each array so that they are accessible on every nodelet. Otherwise, the pointers will be stored on nodelet 0 and will force each thread to frequently migrate back to nodelet 0. This is done by adding the `replicated` keyword, and calling a function to initialize each local copy of the pointer with the value returned from `mw_malloc1dlong` (not shown).

Source Code 4.4: Distributed parallel for loop for striped arrays

```
// Initialized with mw_malloc1dlong(n)
replicated long *a, *b, *c;
for (long i = 0; i < NODELETS() && i < n; ++i) {
    cilk_spawn worker1(&array[i], array, n, grain);
}
cilk_sync;
void worker1(void* hint, long* array, long n, long grain) {
    long stride = grain * NODELETS();
    for (long i = NODE_ID(); i < n; i += stride) {
        long first = i;
        long last = first + stride; if (last > n) { last = n; }
        cilk_spawn worker(array, first, last);
    }
}
void worker2(long* a, long* b, long* c, long begin, long end) {
    for (long i = begin; i < end; i += NODELETS()) {
        c[i] = a[i] + b[i];
    }
}
```

Unfortunately this striped layout is undesirable for many data structures. It fails to take advantage of spatial locality in the array, forcing migrations when elements with nearby indices are accessed together. Furthermore this technique will not work for structure types, which require an arbitrary amount of contiguous memory per element. For this purpose, the

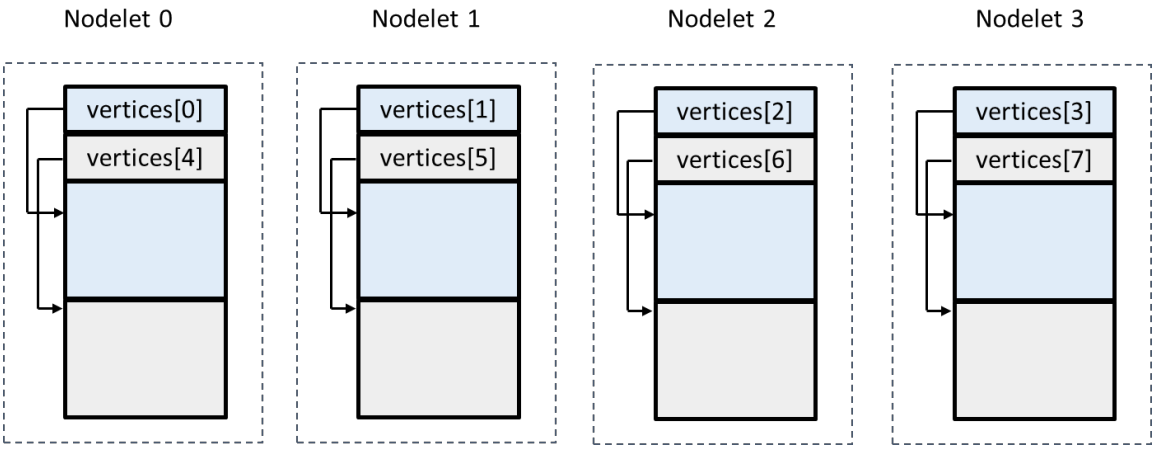


Figure 4.4: Emu memory layout of a two-dimensional **striped** array of vertices.

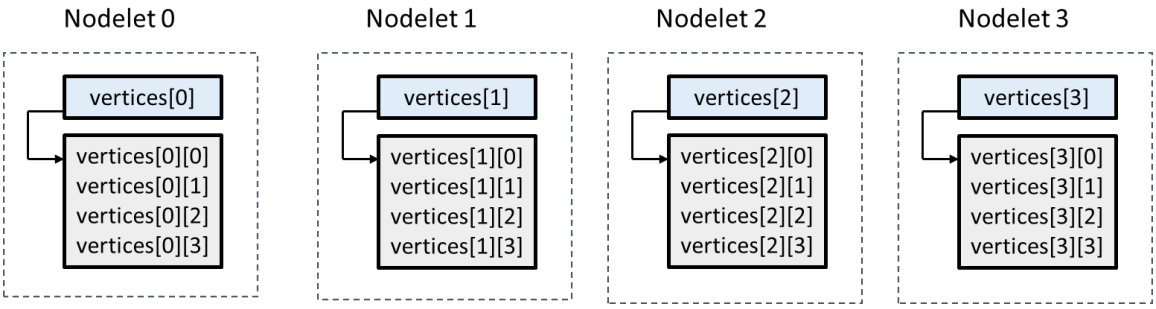


Figure 4.5: Emu memory layout of a two-dimensional **chunked** array of vertices.

Emu library provides `mw_malloc2d`. This function allocates a striped array of pointers to memory chunks of custom size on each nodelet, as depicted in Figure 4.4. Internally, the implementation creates a striped array with `mw_malloc1dlong`, then populates each element with a pointer to a chunk of memory on the same nodelet as the pointer. This function can also be used to create a "chunked" array by requesting one large block on each nodelet, then indexing within each contiguous chunk, as in Figure 4.5. This latter strategy has the advantage of using simple indexing within a single chunk, but complicates the logic required to perform random-access indexes.

Source Code 4.5 implements the STREAM ADD kernel on a chunked array. As in the previous example, a single thread is spawned at each nodelet, which afterwards spawns more worker threads locally. Note that because each invocation of the `worker2` function receives pointers to the chunk on the local nodelet, the loop index here corresponds to

the relative position of the array element within the local chunk, not its absolute position in the global array. This simplification in the indexing logic cannot be used in random access kernels, which must transform all indexing expressions of the form `array[i]` into `array[i/N][i%N]` [1] where N is the chunk size. Besides being more expensive to compute, this also requires the thread to remember the chunk size of each array it accesses.

Source Code 4.5: Distributed parallel for loop for chunked arrays

```
// Initialized with
//    mw_malloc2d(NODELETS(), sizeof(long)*n/NODELETS())
replicated long **a, **b, **c;
for (long i = 0; i < NODELETS(); ++i) {
    cilk_spawn worker1(a[i], b[i], c[i], n/NODELETS(), grain);
}
cilk_sync;
void worker1(long* a, long* b, long* c, long n, long grain) {
    for (long i = 0; i < n; i += grain) {
        long begin = i;
        long end = begin + grain <= n ? begin + grain : n;
        cilk_spawn worker2(begin, end, a, b, c);
    }
}
void worker2(long begin, long end, long* a, long* b, long* c) {
    for (long i = 0; i < end-begin; ++i) {
        c[i] = a[i] + b[i];
    }
}
```

The omission of `cilk_for` makes sense given the diversity of data layouts and thread spawning strategies on this architecture. The compiler cannot determine which of the above

[1] If the chunk size N is a power of 2, this can be transformed into the more efficient expression `data[i >> PRIORITY(N)][i&(N-1)]`

strategies to implement without knowing more about the layout of the arrays accessed within the loop.

4.2.3 Encapsulating data distribution and parallel structure

Managing this complexity calls for a more generic style of programming. Features introduced into modern C++ enable the code in Source Code 4.6, in which the spawning logic is encapsulated into the `parallel_apply` function, the indexing logic is implemented as an overloaded operator, and the operation to apply at each index is passed as a lambda function. This is exactly the approach adopted by the Kokkos [99] library for performance portability. This loop retains the simplicity of the serial for loop from Source Code 4.1 while granting the flexibility to apply arbitrary functions onto diverse data layouts.

In the previous examples, pointers to each array had to be passed down through each worker function. Here, the equals sign in the declaration of the lambda function specifies that `a`, `b`, and `c` should be "captured" by value. Unlike `std::vector`, the `emu_array` class here is designed to perform shallow copies, acting as a handle to the distributed array rather than the array itself. While this automation is convenient, programmers must be careful to note how much state is being implicitly carried within a thread context like this. Each variable or object accessed within the lambda function increases the size of the thread context, which will affect performance when that thread needs to migrate.

Source Code 4.6: Distributed parallel for loop with C++ templates and lambda functions.

```
// Implementation of emu_array<T>
// may use striped or chunked allocation
emu_array<long> a(n), b(n), c(n);
c.parallel_apply([=](long i) {
    c[i] = a[i] + b[i];
});
```

4.3 Benchmarks

Several benchmarks were chosen to characterize the memory performance of the Emu Chick on regular and irregular codes. Determining which of the above data layouts and thread spawn trees is most efficient will be applied to the design of a streaming graph engine in Chapter 4. For each benchmark result, the average memory bandwidth (usually expressed as megabytes per second) is presented.

STREAM: The STREAM [4] benchmark was ported and tuned for the Emu hardware in order to measure raw memory bandwidth. The ADD kernel computes the vector sum of two large arrays of 8-byte integers, storing the result in a third array. On the Emu, these arrays are striped across all the nodelets in the system.

Pointer Chasing: The Pointer Chasing benchmark, which was introduced in section 3.3.3, was also used to evaluate the memory performance of the Emu Chick prototype in the presence of varying levels of spatial locality.

Ping Pong: The simulator validation results in Section 4.4.3 demonstrated a need for a more fine-grained micro-benchmark to illustrate potential differences between hardware and simulated hardware Emu platforms. To explore the cause of this discrepancy, another small benchmark, called "ping pong migration", is introduced. This micro-benchmark measures the bandwidth of thread migrations on the Emu Chick. In each trial, N threads simply migrate back and forth between two nodelets several thousand times.

4.4 Results

4.4.1 STREAM

4.6 shows the results from running the STREAM benchmark on a single Emu nodelet. Performance scales up with thread count until 32, after which it levels off at 120 MB/s. Recall that in the `serial_spawn` strategy, a single thread uses a for loop to create each worker thread, while `recursive_spawn` uses a recursive spawn tree to create the threads. There

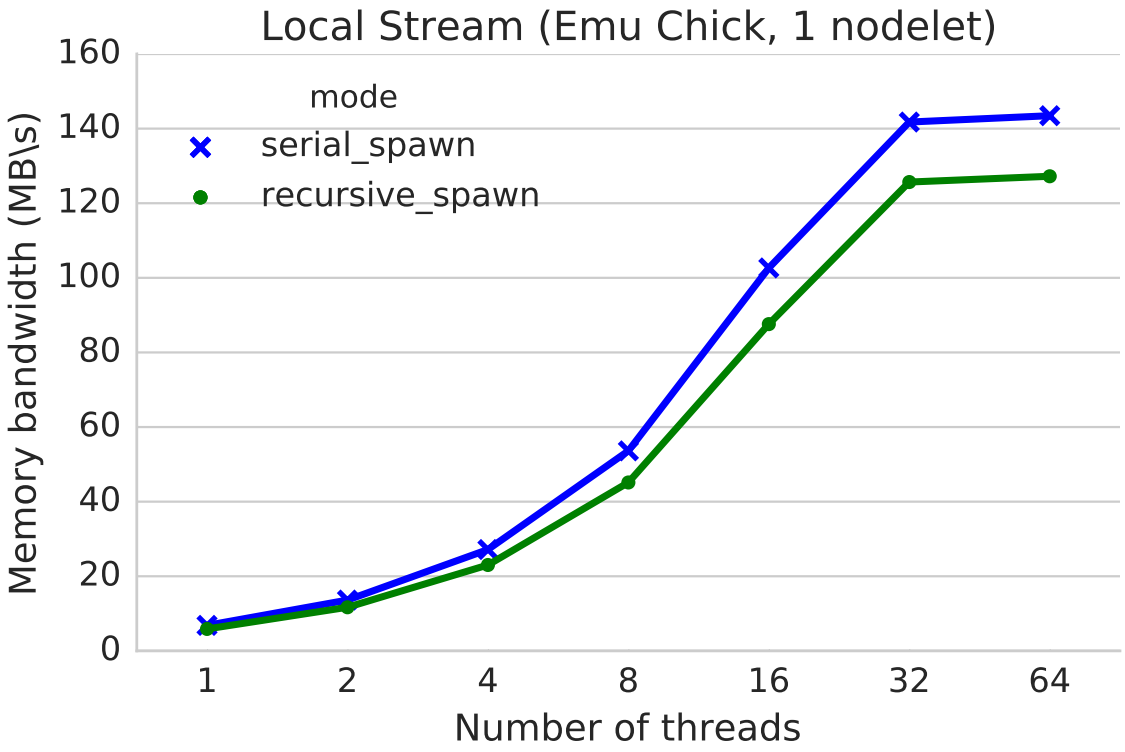


Figure 4.6: Memory bandwidth achieved on a single node of the Emu Chick. Threads are created using a serial loop or a recursive spawn tree.

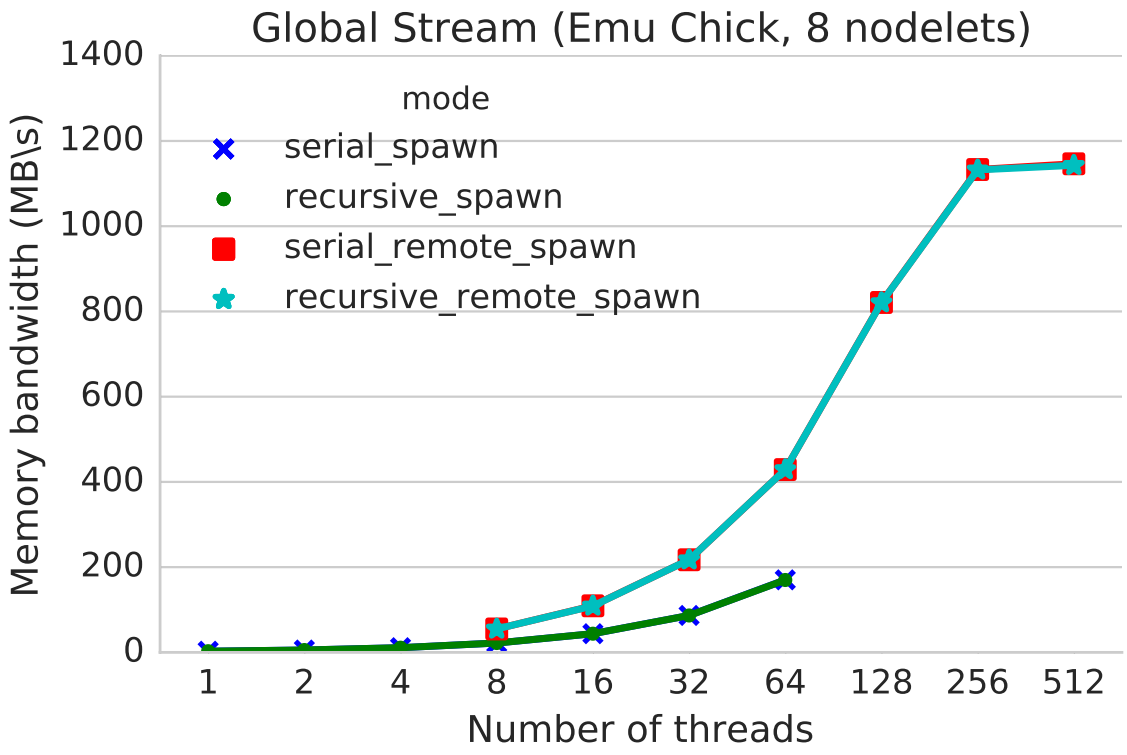


Figure 4.7: Memory bandwidth achieved on eight nodes of the Emu Chick. The remote spawn variants create a thread on each nodelet which subsequently creates the local worker threads.

is not much difference between the two approaches, indicating that thread creation is not the bottleneck within a single nodelet.

In 4.7, the STREAM benchmark is extended to run on eight nodelets (one node) of the Emu Chick. Two new thread creation strategies are introduced here, `serial_remote_spawn` and `recursive_remote_spawn`. A remote spawn on Emu means that the thread is created on a remote nodelet, rather than being created locally and allowed to migrate to the remote data. The remote thread creation strategies first create a thread on each nodelet (either one at a time or with a recursive spawn tree), and then perform a second level of spawning on the local nodelet, as in the single nodelet case. The results show that remote spawns are essential to achieving maximum bandwidth on Emu.

The reference Xeon system achieves close to the nominal bandwidth of 51.2 GB/s on the STREAM benchmark. In comparison the Emu Chick is still rather slow, reaching only

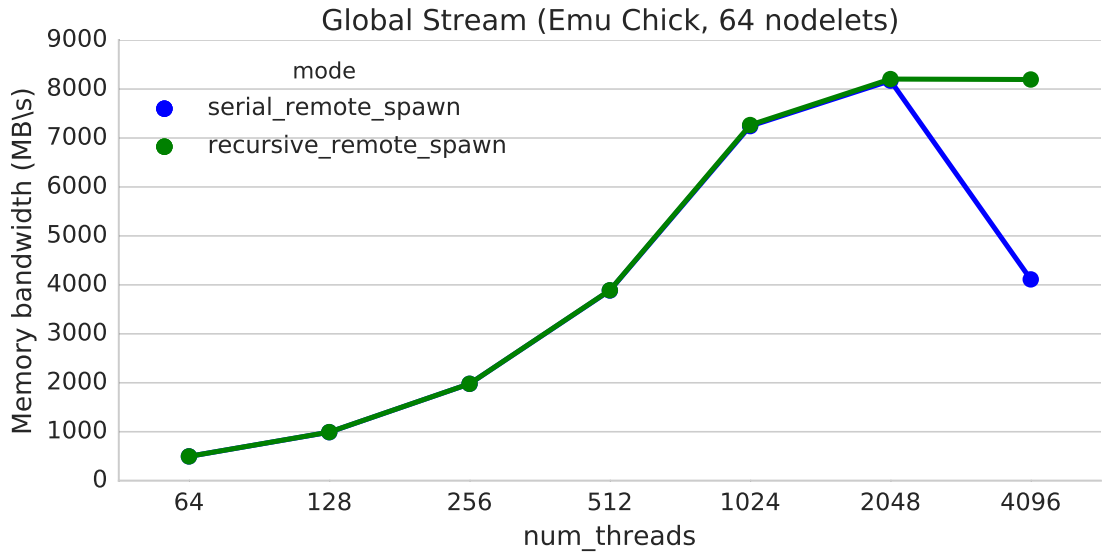


Figure 4.8: Memory bandwidth achieved on 8 nodes of the Emu Chick. The multi-node configuration was only stable enough to collect results for the serial remote spawn variant.

1.2 GB/s on a single node card while the full 8-node configuration of the Emu Chick yielded 8 GB/s (figure 4.8). However even in its current state this prototype system demonstrates improvements in other benchmarks where the memory access pattern is not linear and predictable as it is in STREAM.

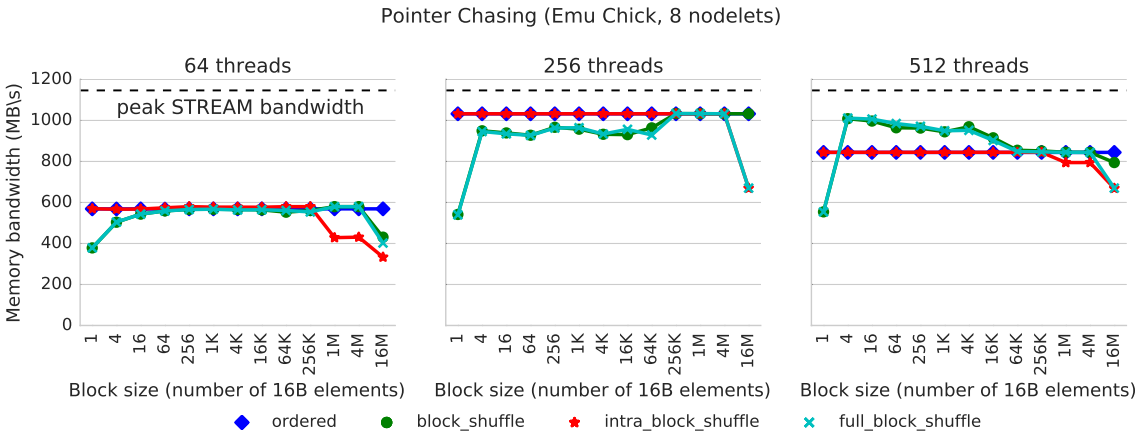


Figure 4.9: Pointer chasing performance on a single node of the Emu Chick.

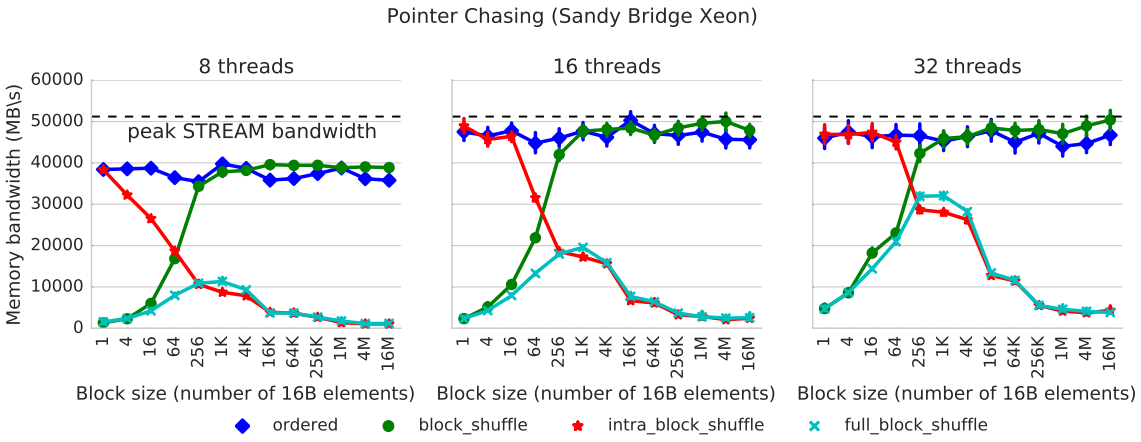


Figure 4.10: Pointer chasing performance on Xeon

4.4.2 Pointer Chasing

Figures 4.9 and 4.10 compare the performance of the Emu Chick against the Xeon server system for the pointer chasing benchmark. These results reveal important characteristics of both systems and highlight the unique advantages of the Emu Chick.

Pointer chasing on the Xeon architecture performs poorly for several reasons. For small block sizes, the memory system bandwidth is used inefficiently. An entire 64-byte cache line must be transferred from memory, but only 16 bytes will be used. The best performance is achieved with a block size between 256 and 4K elements. This corresponds to a memory chunk of about 8KB, the size of one DRAM page. Regardless of the size of the access, an entire DRAM row must be activated for each element traversed. Adding more threads at this point increases the number of simultaneous row activations. As the block size grows beyond the size of a DRAM page, performance declines again.

With two exceptions, performance on Emu remains stable regardless of block size. Emu's memory access granularity is 8 bytes, so it never transfers unused data in this benchmark. As long as a block fits within a single nodelet's local memory channel, there is no penalty for random access within the block. The block size of 1 is an interesting case: here Emu threads are likely to migrate on every access, and so performance is greatly reduced. But performance recovers when even as few as four elements are accessed between each

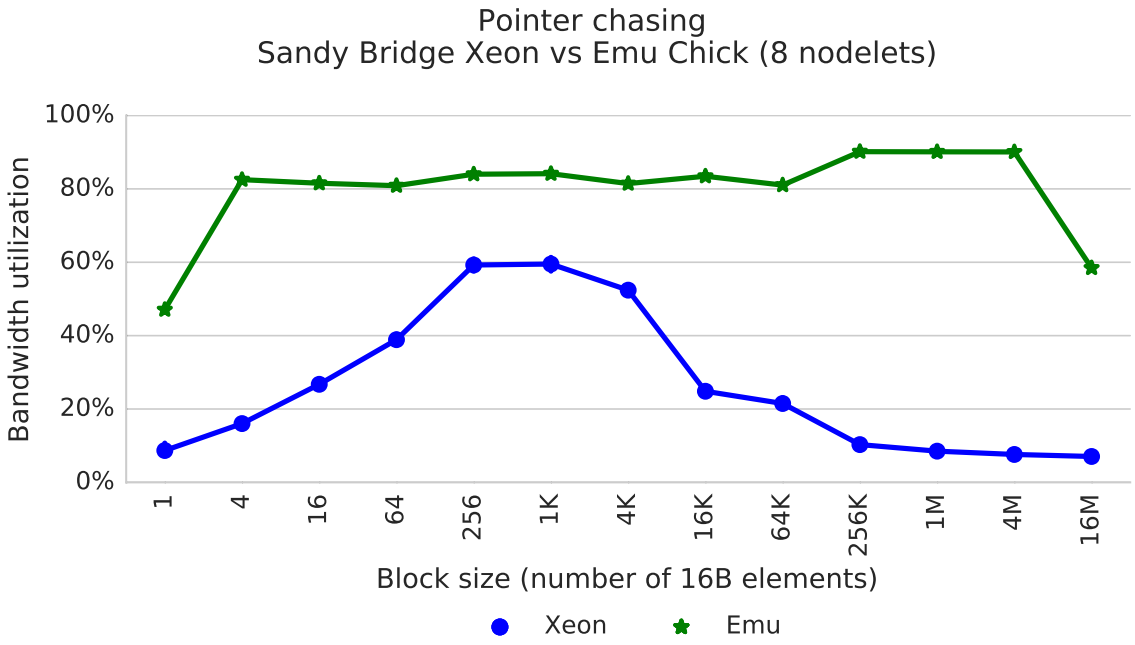


Figure 4.11: Bandwidth utilization of pointer chasing, compared between Xeon and Emu

migration. A similar trend is seen when the entire array is shuffled as a single block.

In 4.11 the performance of each system has been normalized to the total bandwidth of the system (i.e. the best result on the STREAM benchmark). In the pointer chasing benchmark, the Emu system is much better at using the available system bandwidth, using 80% of available system bandwidth in most cases and 50% in the worst cases. Xeon uses less than 25% bandwidth in most cases, relying on multi-kilobyte levels of locality to efficiently transfer the data. These results bode well both for the targeted streaming graph and tensor decomposition applications which have pointer chasing behavior and rely on random accesses to compute SpMV and SpMM operations, respectively.

4.4.3 Simulator Validation

Section 4.4.4 will predict the performance of an Emu Chick system operating at full speed as well as larger configurations by using the provided Emu simulator. Before attempting this, the simulator was validated by configuring it to match the specifications of the current hardware system. The results of this evaluation are displayed in Figure 4.12. While the

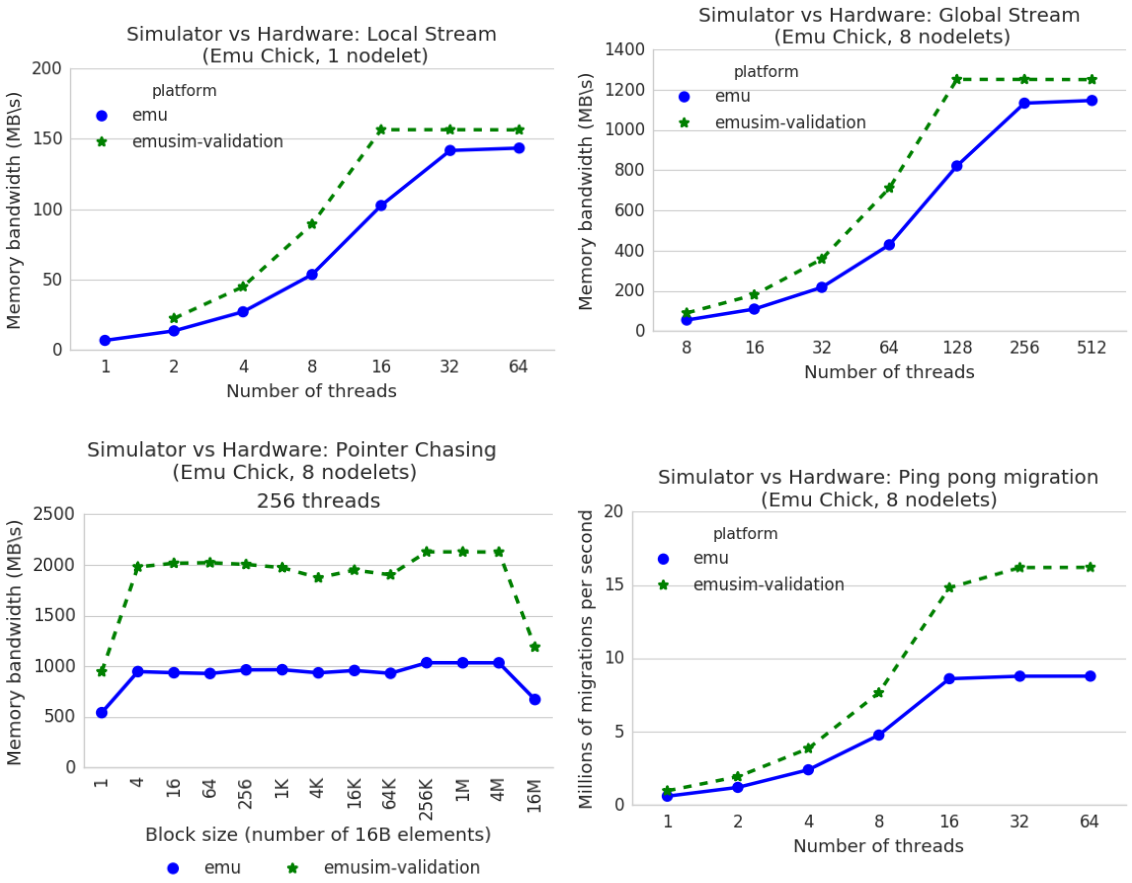


Figure 4.12: Emu hardware performance compared with simulator results

STREAM benchmark results match well for both single nodelet and multi-nodelet operation, the pointer chase benchmark results do not. Despite the error in magnitude, the shape of the results matches well.

To help explain this difference, Figure 4.12 also shows results from the hardware and simulated ping pong benchmark. While the simulator can perform 16 million migrations per second, the hardware is currently limited to only 9 million migrations per second. Since pointer chasing is a migration-heavy benchmark, the performance of the thread migration engine affects its performance to a much greater degree than STREAM. These experiments indicate that the latency for a single thread migration on the current system is approximately 1-2 μs.

4.4.4 Extrapolation to future systems

Since the simulator has been shown to closely match the hardware for STREAM results, the results from scaled-up configurations are likely to be accurate.

When the STREAM results are extended to full speed in the simulator, the need for remote spawns becomes even more apparent. Figure 4.13 shows the bandwidth on a single node scaling up to 10 GB/s for serial and remote spawn variants. Unlike the hardware results, which reached peak bandwidth with only half of the thread capacity, the simulator predicts that all 2048 threads are required to achieve peak bandwidth on a single node for this benchmark. These trends become even more pronounced in the results for a full-speed 8-node Emu Chick system, depicted in Figure 4.14. The local spawn variants never attain more than 10 GB/s due to congestion while all threads migrate away from node 0. Even with 64 nodelets, the serial remote spawn is able to cover the entire system with threads just as well as the recursive remote spawn. It may be that nodelet-level recursive spawns will only become necessary once the system is scaled up to hundreds of nodes, as in the rack mount system mentioned in Table 4.1.

Finally, 4.15 plots simulation results for the full-speed configuration of an 8 node Emu

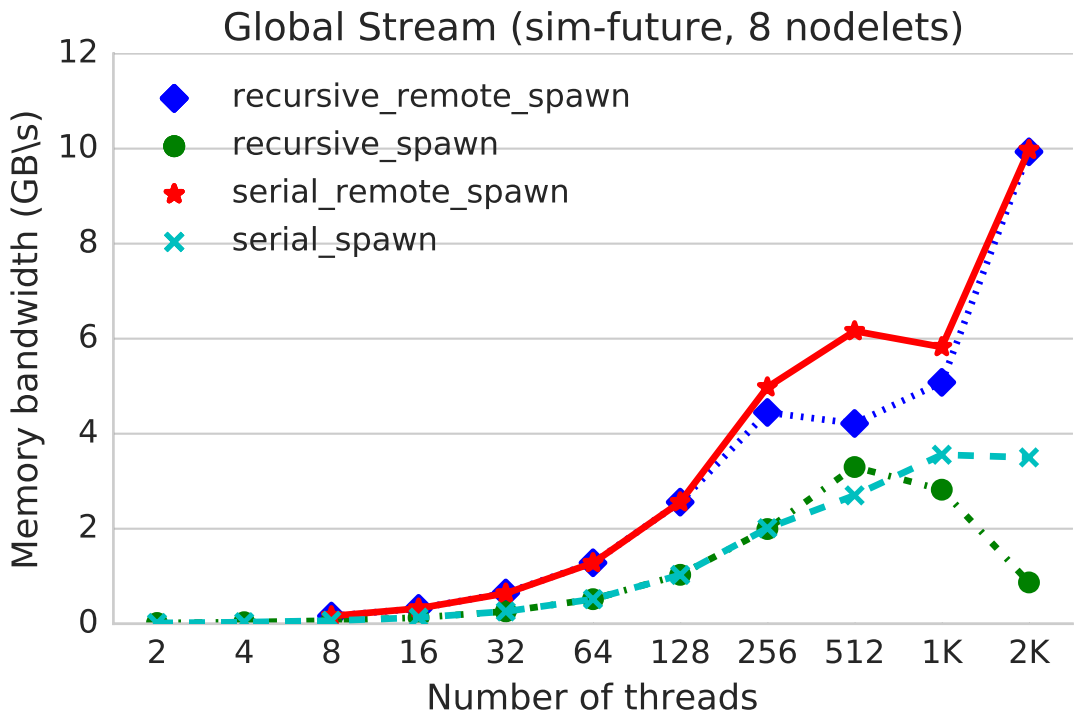


Figure 4.13: Simulator results for the STREAM benchmark running on a single Emu node

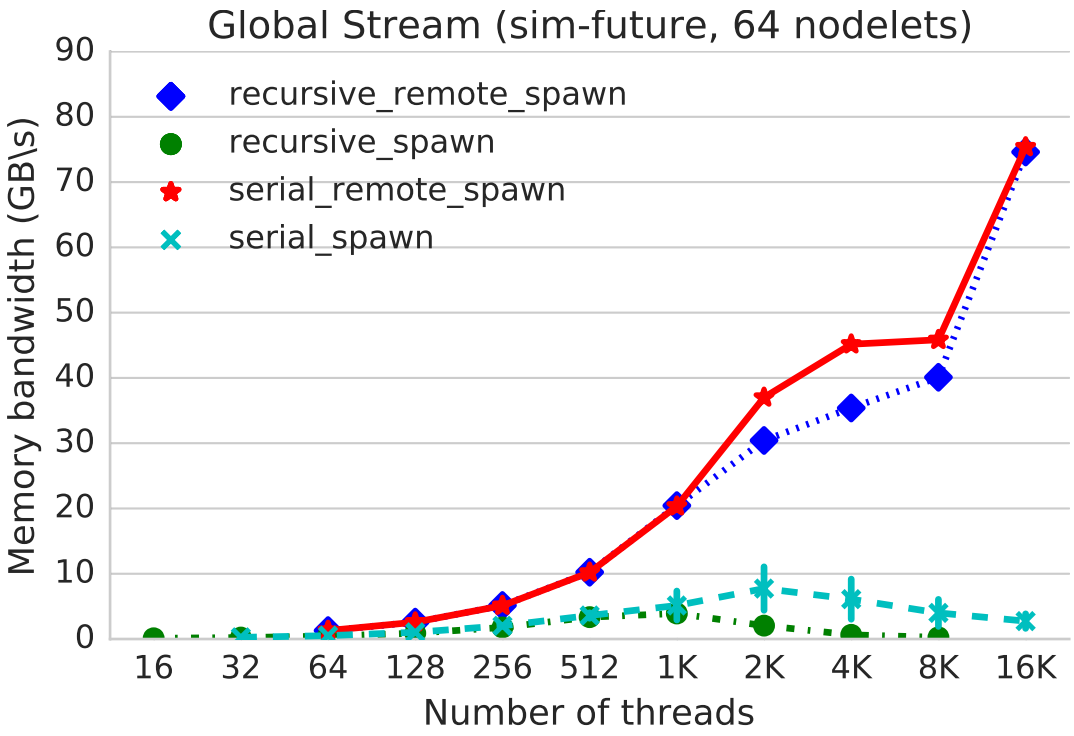


Figure 4.14: Simulator results for the STREAM benchmark running on the Emu Chick

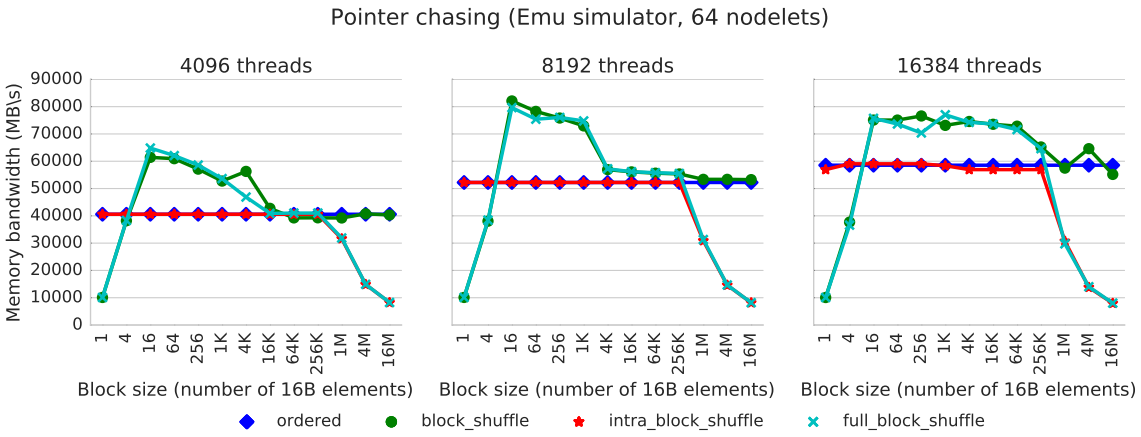


Figure 4.15: Simulated results for the pointer chasing benchmark running on an Emu Chick at full speed.

system. Despite the increase in scale, the system performance is still not sensitive to the granularity of spatial locality, and scales well even up to thousands of threads. This chart predicts that the full-speed Emu Chick hardware will exceed the performance of the baseline Xeon server on the pointer chasing benchmark.

4.5 Conclusion

The initial evaluation of the Emu Chick demonstrates some of the limitations of the existing prototype system as well as some potential benefits for massive data analytics applications like streaming graph analytics and sparse tensor decomposition.

Initial results demonstrate low overall bandwidth for the Emu system but illustrate that it can achieve a high percentage of effective memory bandwidth even in a worst-case access scenario like pointer chasing, which achieves a stable 80% bandwidth utilization across a wide range of locality parameters.

These results and initial results on how data layouts can improve random access provide a template for future benchmarking and application development and show how application memory layouts and "smart" thread migration can be used to maximize performance on the Emu system.

CHAPTER 5

OPTIMIZING STREAMING GRAPH ANALYTICS FOR THE EMU CHICK

This chapter applies the knowledge gleaned from the experiments in Chapter 4 to develop a dynamic graph data structure and streaming graph algorithms for the Emu Chick.

5.1 Graph Data Structures

A new in-memory graph engine, named MeatBee, was written specifically for the Emu Chick prototype. The source code is inspired by STINGER rather than CSR to enable future work with streaming data and incremental algorithms, one of the primary targets of the Emu architecture.

The data structure design of MeatBee is illustrated in Figure 5.1. The vertex array is striped across all nodelets in the system, such that vertex 0 is on nodelet 0, vertex 1 is on nodelet 1, and so on. Unlike STINGER, the neighbor list of each vertex is implemented as a hash table with a small number of buckets. Each bucket is a pointer to a linked-list of edge blocks, each of which stores a fixed number of adjacent vertex IDs and a pointer to the next edge block. The neighbors of a single vertex can be updated and traversed in parallel by multiple threads by spawning a thread at each bucket. MeatBee can be configured to store in-edges for efficient traversal of the graph in reverse, but it can also be configured to save capacity by only storing out-edges. MeatBee supports an arbitrary number of edge properties, configured at compile-time. For the experiments in this section, MeatBee was configured to store a directed graph (out-edges only), with two edge properties (weight and timestamp), and one bucket per vertex.

To avoid the overhead of generic run-time memory allocation via `malloc`, each nodelet pre-allocates a local pool of edge blocks. A vertex can claim edge blocks from any pool, but it is desirable to string together edge blocks from the same pool to avoid thread migra-

tions during edge list traversal. When the local pool is exhausted, the edge block allocator automatically moves to the pool on the next nodelet.

5.2 Algorithms

5.2.1 Graph Construction

Kernel 1 of the Graph500 benchmark involves constructing a graph data structure from a list of edges. DynoGraph further requires incremental graph construction. MeatBee uses the same algorithm to implement both tasks.

The batch insert algorithm from Section 3.1 was ported to work for the Emu architecture. Since the algorithm relies on quickly pre-sorting the list of edges, a parallel merge sort algorithm for Cilk Plus [100] was ported and tuned for the Emu architecture, although radix sort [101] has also been shown to work well on this architecture. Besides converting from OpenMP to Cilk, several important changes were introduced to tune the algorithm for the Emu Chick.

Recall that each thread is given a subrange of edge updates to apply as it traverses the edge list. On Emu, the list of edges is loaded from disk into memory on nodelet 0. The subranges of each thread constitute non-overlapping views into this buffer. When an Emu thread migrates to update an edge list on a remote nodelet, the edges within the sub-range remain on nodelet 0. In this case the thread would be forced to migrate rapidly back and forth between the subrange and the edge list. This would severely limit performance and cause nodelet 0 to become a bottleneck.

An additional step was added to the beginning of the algorithm in Source Code 3.1. During the initial sorting step, nodelet 0 sorts the list to group together edges that will apply to the same nodelet. Because the vertex array is striped across all nodelets, this can be done by defining the sort key as the low bits of the source vertex ID of each edge. Once the sorting is complete, nodelet 0 scatters the list across all the nodelets using remote writes. Once the list has been scattered, threads are remote-spawned at each nodelet to sort

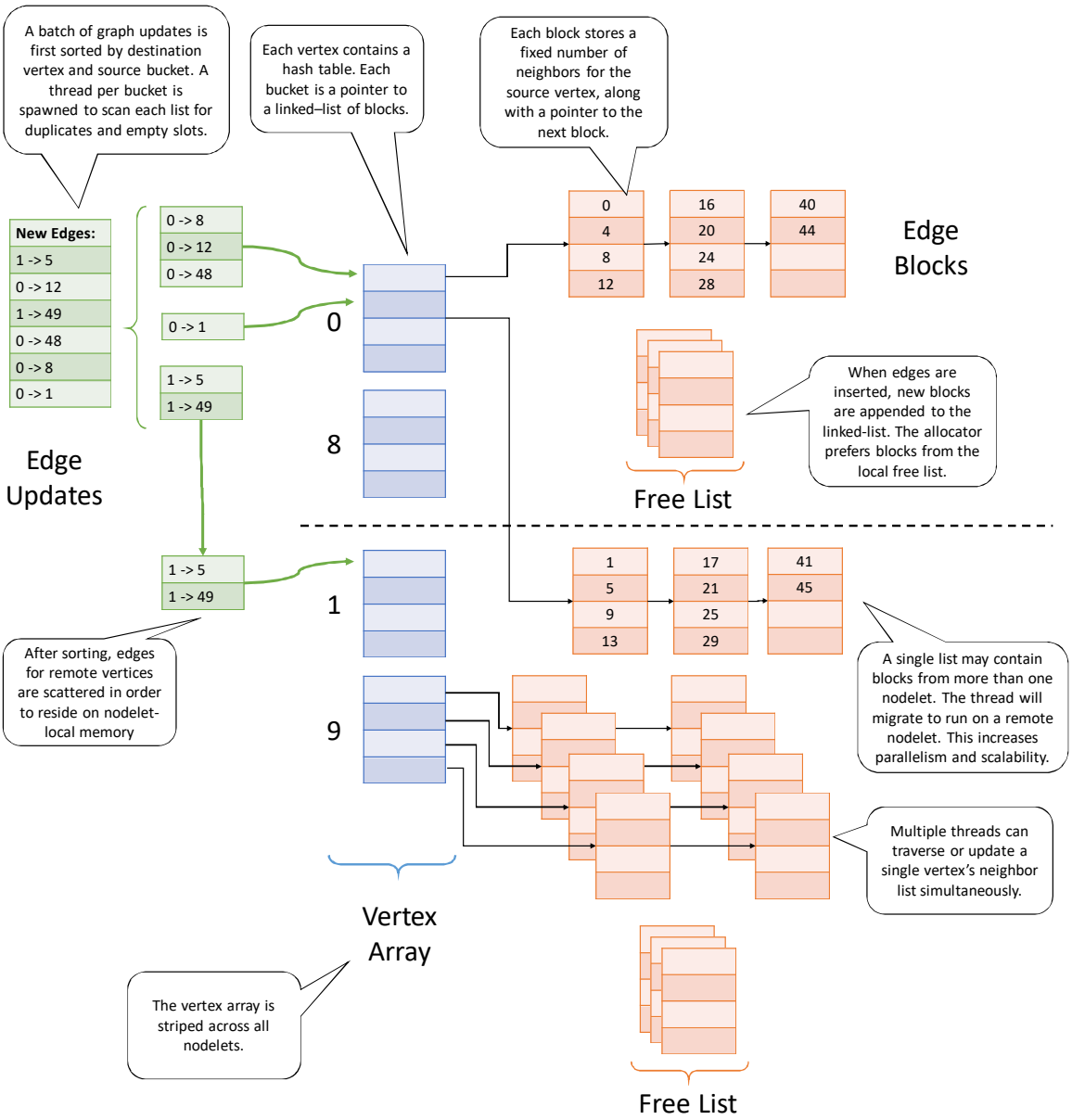


Figure 5.1: MeatBee graph data structure layout

the local list of updates by source vertex and destination bucket. From here the algorithm continues as before, splitting the local list of updates into sub-ranges and spawning threads to handle each range.

This highlights an important design challenge for algorithms on Emu. The intent of Emu's migrating thread paradigm is that the "function" migrates to the "data". In practice the distinction between these two categories is not always clear. The small list of edge updates is being treated as static data, but for optimal performance it should migrate automatically with the worker thread. Statically copying each edge to a remote nodelet works with the current static graph partition, but will fail to interact well with dynamic graph partitioning schemes.

5.2.2 Breadth-first Search

Source Code 5.1: BFS algorithm using migrating threads

```
for v in range(num_vertices):
  parent[v] = -1
queue.push(root)
while len(queue) > 0:
  for src in queue:
    for dst in out_edges(src):
      # Thread migrates here
      if parent[dst] == -1:
        if compare_and_swap(parent[dst], -1, src):
          queue.push(dst)
```

The initial implementation of Breadth-first search (Source Code 5.1) was a direct port of the STINGER code. Each vertex iterates through each of its neighbors and tries to set itself as the parent of that vertex using an atomic compare-and-swap operation. If the operation is successful, the neighbor vertex is added to the queue to be explored along with the next

frontier.

On Emu, the parent array is striped across nodelets in the same way as the vertex array. Each nodelet contains a local queue so that threads can push vertices into the queue without migrating. At the beginning of each frontier, threads are spawned at each nodelet to explore the local queues. Thread migrations do occur whenever a thread attempts to claim a vertex that is located on a remote nodelet. In the common case, a thread reads an edge, migrates to the nodelet that owns the destination vertex, executes a compare-and-swap on the parent array, pushes into the local queue, and then migrates back to read the next edge. If the destination vertex happens to be local, no migration will occur when processing that edge.

Source Code 5.2: BFS algorithm using remote writes

```
for v in range(num_vertices):
  parent[v] = -1
  new_parent[v] = -1
queue.push(root)
while len(queue) > 0:
  for src in queue:
    for dst in out_edges(src):
      # Remote write
      new_parent[dst] = src

  for v in range(num_vertices):
    if parent[v] == -1:
      if new_parent[v] != -1:
        parent[v] = new_parent[v]
        queue.push(v)
```

An alternative BFS implementation (Source Code 5.2) was developed to exploit the capability of the Emu system to efficiently perform remote writes. A copy of the parent array (`new_parent`) was introduced to hold intermediate state during each frontier. Now,

rather than migrating to the nodelet that contains the destination vertex, the thread issues a remote write on the `new_parent` array. The remote write packet can travel through the network and complete asynchronously while the thread that created it continues to traverse the edge list. Remote writes attempting to claim the same vertex are serialized in the memory front end of the remote nodelet. Rather than attempting to synchronize these writes, later writes simply overwrite earlier ones. The Graph500 specification [45] allows this benign race condition, which also exists in the previous algorithm when two threads try to claim the same vertex as a child. After all the remote writes have completed, the second step scans through the `new_parent` array looking for vertices that did not have a parent at the beginning of this frontier (`parent[v] == -1`) but were assigned a parent in this iteration (`new_parent[v] != -1`). When such a vertex is found, it is added to the local queue, and the new parent value `new_parent[v]` is copied into the parent array at `parent[v]`. This is similar to direction-optimizing BFS [64] and may be able to adopt its early termination optimizations.

5.3 Experimental Setup

5.3.1 Emu Simulator

Emu provides a cycle-accurate simulator along with the compiler toolchain to aid in testing and evaluating software before running on the hardware. The simulator counts key performance events such as the number of thread spawns, migrations, and memory operations per nodelet. Previous work [102] has shown that while the simulator underestimates the effects of thread migration costs (1-2 μs on the hardware), performance curves in the simulator track relatively closely to the actual hardware. This study uses the simulator to project performance to multi-node execution since multi-node applications are not currently able to run on the prototype hardware. Simulation of single-threaded execution can run at 1000x slowdown but more complex simulations can be much slower, so simulations with large input graphs are not able to be run in a reasonable amount of time with the simulator.

5.3.2 Experiment Configurations

All experiments are run using Emu's 18.02 compiler and simulator toolchain, and the Emu Chick system is running NCDIMM firmware version 2.1.7, system software version 1.4, and each stationary core is running the 2.0.32 version of software. Results are presented for several configurations of the Emu system:

- Emu Chick hardware **(HW)**: single-node (8 nodelets) due to aforementioned firmware limitations. All hardware results are reported for one Emu Chick node.
- Emu Chick simulator-current **(Sim-current)**: matches the current hardware configuration with 1 GC per nodelet clocked at 150 MHz.
- Emu Chick simulator-future **(Sim-future)**: matches the planned future hardware configuration with 4 GCs per nodelet clocked at 300 MHz.

The primary metric for comparison across algorithms is memory bandwidth (MB/s) and effective memory bandwidth utilization (% of measured peak memory bandwidth). This metric is picked due to the difficulty in comparing the near-data processing and NCDIMM configuration of the Emu with traditional CPU-based systems. Previous investigations [102] have shown that the Emu hardware can achieve up to 1.2 GB/s per node and 6.5 GB/s on 8 nodes for the STREAM benchmark, which is used as the "peak" memory bandwidth number. Traversed Edges Per Second (TEPS) and scale size are also reported for the Graph500 BFS since this is a standard and highly recognizable metric for this application.

BFS uses uses RMAT graphs as specified by Graph500 [45] and uniform random (Erdös-Renyi) graphs [103], scale 10 through 17, from a generator in the STINGER code base.

5.3.3 Mini-DynoGraph

Having been selected to benchmark high-end shared-memory server systems, the Dyno-Graph input graphs introduced in Section 2.5 are too large to run on the Emu Chick hard-

Table 5.1: Miniaturized DynoGraph input graph sizes

	Description	# of Vertices	Total Edges	Unique Edges	Edge Factor
sc15	NetFlow data from SCinet 2015	32 K	524 K	85 K	4.73
dns2	Passive DNS	32 K	1 M	32 K	1.40
worldcup	Twitter data from the 2014 World Cup	32 K	221 K	34 K	2.58
RMAT	RMAT scale 15	32 K	512 K	512 K	16

ware prototype, and much too large to run on the Emu simulator. In order to perform a preliminary evaluation of the system, the inputs were scaled down by sampling the first 32K vertices. The sc15 graph was additionally truncated to reduce the total number of edges. The resulting graph sizes are listed in Table 5.1.

Additionally, the scale-15 RMAT graph was permuted to simulate a burst of edges to a high degree vertex. This was done by shifting all of the edges connected to the highest-degree vertex in the graph to the end of the edge list.

5.4 Results

5.4.1 Graph500

Figure 5.2 compares the two BFS algorithms running on the hardware. The migrating threads implementation is initially more efficient, since it does not need to scan all vertices for changes between each frontier. The remote write algorithm is more scalable as the graph size increases. This indicates that in the current prototype, an Emu nodelet can handle a large number of incoming remote writes more efficiently than it can handle a deluge of incoming thread migrations. Figure 5.3 extends these results to an 8-node configuration of sim-future to confirm that the performance of the remote write algorithm scales better as the size of the system increases. The remaining BFS result plots will all use the remote writes algorithm.

The initial graph engine implementation does not attempt to evenly partition the graph across the nodelets in the system. The neighbor list of each vertex is co-located with the

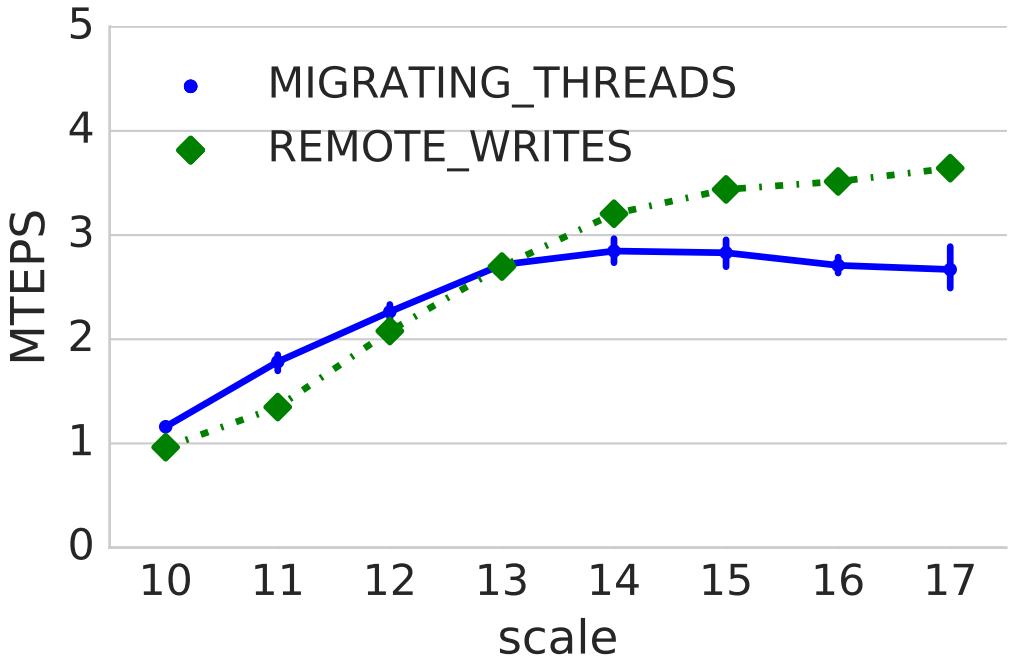


Figure 5.2: Graph500 BFS benchmark results on HW, comparing the scalability of the migrating threads BFS algorithm against the remote writes BFS algorithm.

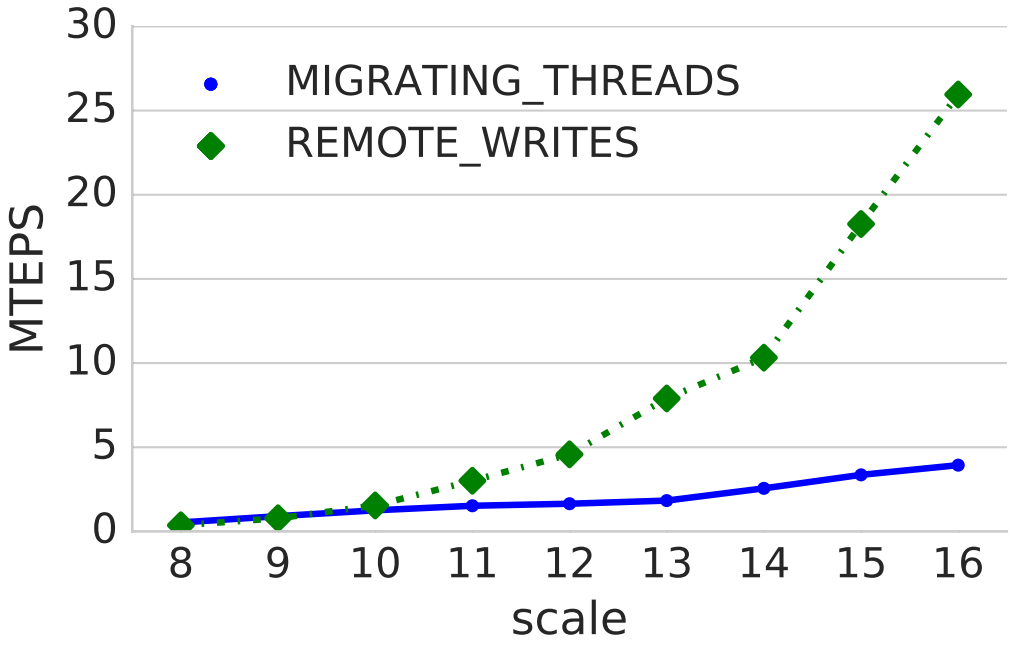


Figure 5.3: Graph500 BFS benchmark results on a 64-nodelet configuration of sim-future, demonstrating the superior scalability of the BFS algorithm using remote writes.

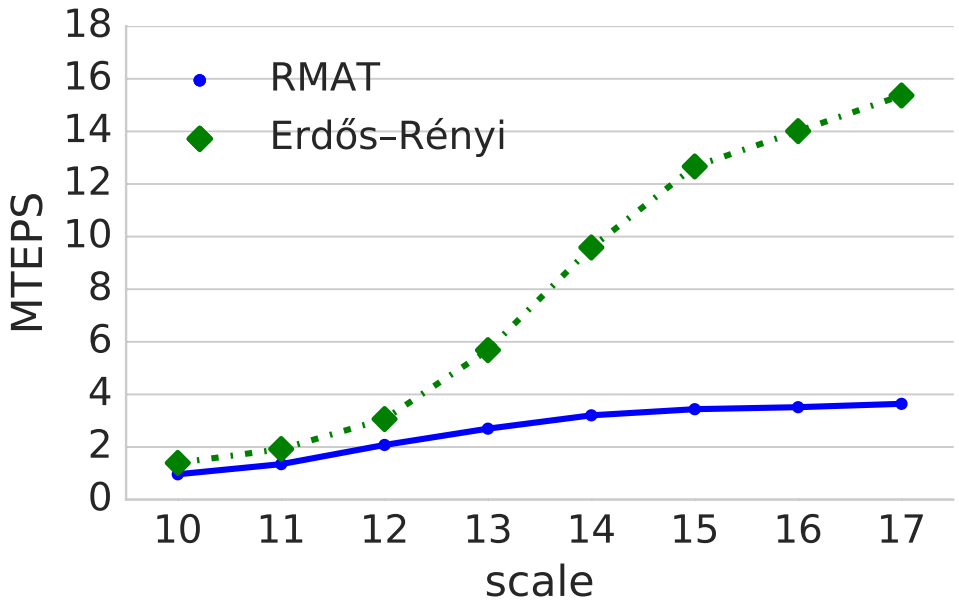


Figure 5.4: Graph500 BFS benchmark results on HW, demonstrating the importance of a balanced graph distribution.

vertex on a single nodelet. The RMAT graphs specified by Graph500 have highly skewed degree distributions, leading to uneven work distribution for BFS. Figure 5.4 shows that running the benchmarks with balanced ErdösRényi graphs instead leads to better performance. Future work will enhance the graph construction algorithm to create a better partition for power-law graphs.

Figures 5.5 and 5.6 show the BFS simulation results for single and multi-node configurations of sim-current and sim-future up to scale 16 graphs. A peak performance of 26 MTEPS for RMAT graphs and 47 MTEPS for ErdösRényi graphs is achieved. Based on the performance of 8-nodelet sim-future, one would expect the peak multi-node performance to be much higher. Note that while a scale 16 graph (64K vertices) was the largest that could run in the simulator, it may still be too small to keep all 16K threads of the multi-node system busy. Other factors that limit scaling in the multi-node results are discussed in Section 5.5.

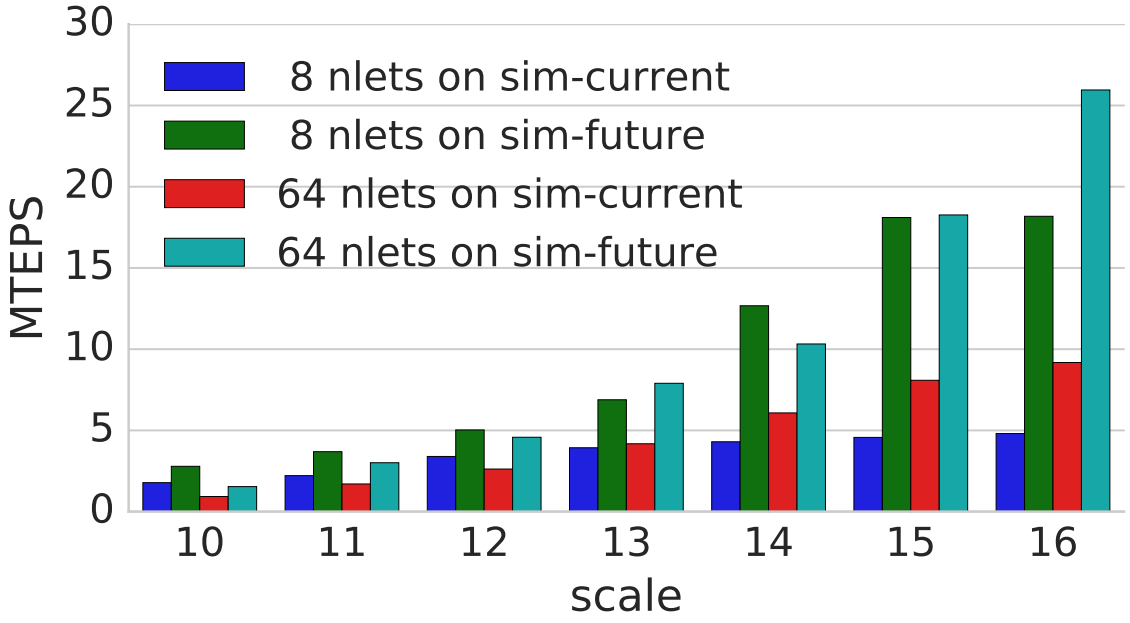


Figure 5.5: Graph500 BFS benchmark results on simulator with **unbalanced** (RMAT) graphs, demonstrating performance scalability to future hardware.

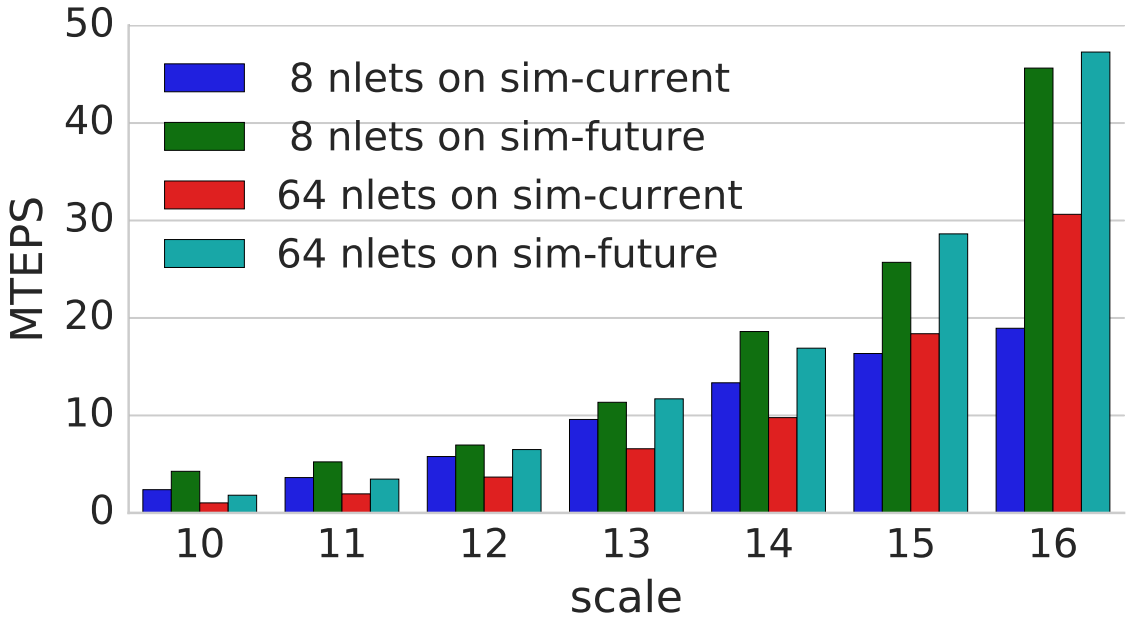


Figure 5.6: Graph500 BFS benchmark results on simulator with **balanced** (Erdös-Rényi) graphs, demonstrating performance scalability to future hardware.

5.4.2 DynoGraph

Figure 5.7 plots the time taken to insert each batch for the reduced-size DynoGraph input graphs. These results mirror those in Figure 5.7. Once again, the RMAT graph has far less variance between batches than the real-world inputs. Overall the MeatBee implementation is able to avoid catastrophic slowdowns in the rate of graph construction as the graph grows. Furthermore the sim-future configuration promises to improve performance by 2-3x.

Figure 5.8 compares the effect of varying the batch size when inserting RMAT graphs. As in the STINGER implementation, larger batches allow for more parallelization during the edge insertion process to amortize startup costs. In these simulations, scaling up to a 64-node system does not show any performance improvement. A scale-15 graph is rather small to justify a distributed system, much of the benefit of the additional processing power is overshadowed by the additional migration overhead and coordination time. Larger graph inputs may display better scaling performance.

The RMAT graph was permuted in order to simulate a burst of updates to a high-degree vertex after the graph had been constructed. Figure 5.9 compares the permuted with the un-permuted RMAT graph. As expected, the time to insert the last few batches in the benchmark rises sharply in the permuted graph. This suggests a technique for extending DynoGraph with a synthetic graph generator to stress-test an incremental graph construction algorithm. The worst-case batch time determines the rate at which a streaming graph application can ingest edge updates.

5.5 Discussion

Besides those presented here, several other graph algorithms were ported to the Emu architecture, including Single Source Shortest Path (SSSP), Triangle Counting, and PageRank. These algorithms require further analysis for the Emu architecture's unique requirements as well as architectural fixes. The issues encountered are briefly summarized here:

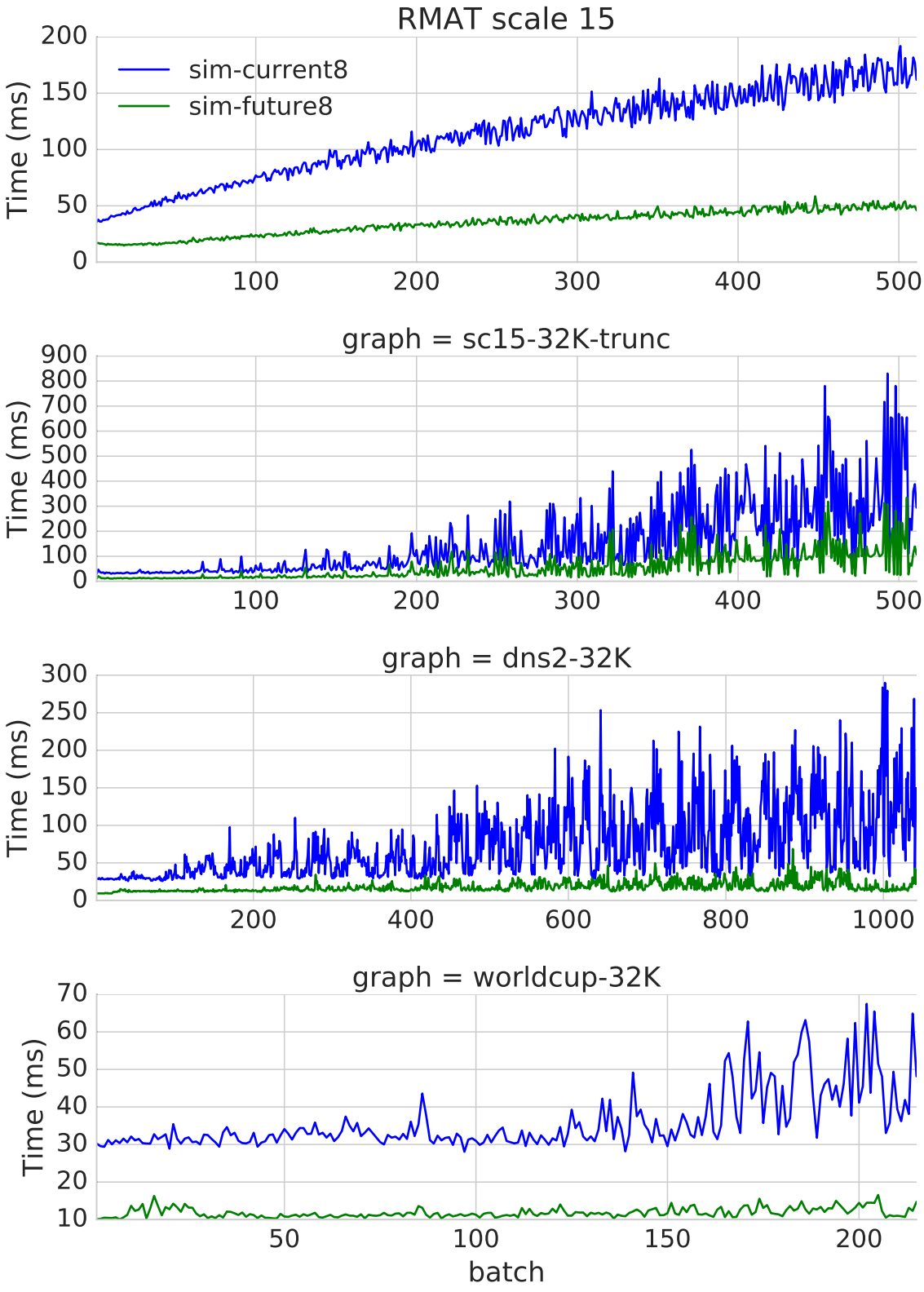


Figure 5.7: Simulation results: time taken to do streaming insertions for mini-DynoGraph inputs.

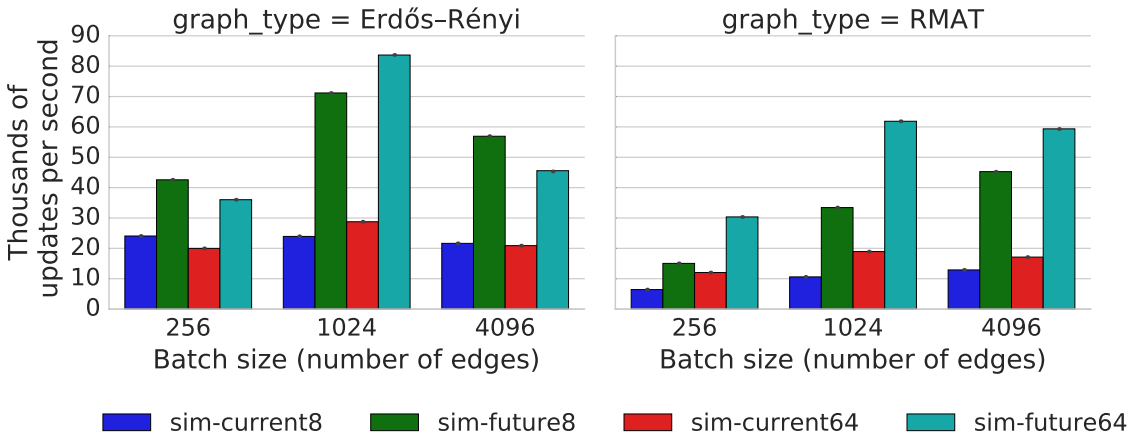


Figure 5.8: Simulation results: Average edge insertion rate for scale 15 graphs.

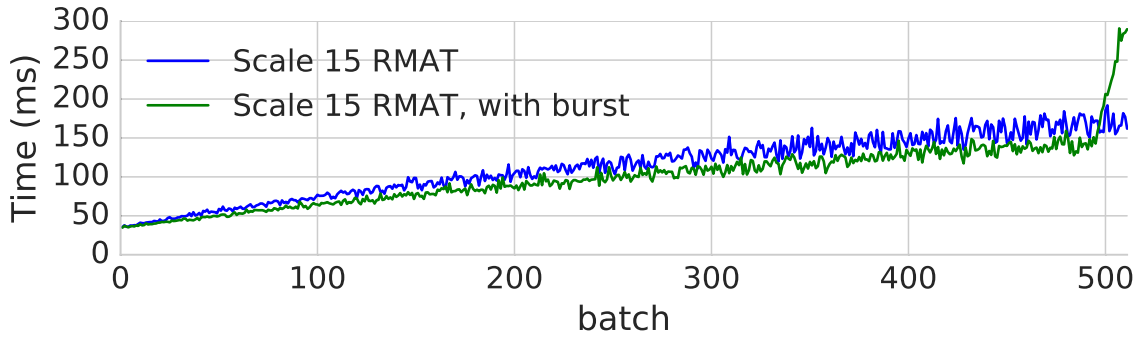


Figure 5.9: Result of moving the highest-degree vertex to the end of the stream of RMAT edges, to simulate a burst of updates to a high-degree vertex. Notice the spike in the time taken insert the last few batches.

- On each step of the Bellman-Ford SSSP algorithm [104], each vertex updates its neighbors with the new shortest distance between them. Emu's remote minimum atomic could be used here in a similar way to the remote-write BFS described above. However a hardware bug in the remote minimum atomic prevented these results from being collected.

- PageRank can be implemented using either a pull-based or push-based algorithm [105]. The latter case could use remote atomic adds to sum up the contributions from each vertex. Unfortunately PageRank requires floating-point arithmetic, and Emu only supports integer remote atomics at this time.

- Triangle Counting is expressed as three nested edge list traversals: For each edge $i \rightarrow j$, for each edge $i \rightarrow k$, look for an edge $j \rightarrow k$, where $i < j < k$ to avoid counting triangles more than once. Remembering the position of all three edge list traversals simultaneously exerts significant register pressure on this workload, preventing worker threads from migrating efficiently.

From these algorithm implementations, five performance-limiting of the Emu Chick have been identified: 1) **Thread stack placement** and remote thread migrations back to a "home" nodelet that contains the thread stack. 2) **Thread spawn limits** are currently not tracked by the hardware and unbounded spawns can crash the hardware or cause thread migration "hotspots." 3) It is tough to get proper **workload balance** when using irregular data structures like unbalanced graphs. 4) Following from (3), input sizes are limited by the need to create **distributed data structures** from an initial chunk of data on the "home" node. 5) The Emu is a **non-uniform PGAS** system but with variable costs for remote "put" and "get" operations. Below, each of these issues is addressed in more detail in relation to the evaluated algorithms.

Thread Stack Placement: A stack frame is allocated on a nodelet when a new thread is spawned. Threads carry their registers with them when they migrate, but stack accesses

require a migration back to the originating nodelet. If a thread needs to access its stack while manipulating remote data, it will migrate back and forth in a ping-pong fashion. The usage of thread stacks and ping-pong migration can be prevented by obeying the following rules when writing a function that is expected to migrate: 1) Maximize the use of inlined function calls because normal function calls require a migration back to the home nodelet to save the register set. 2) Write lightweight worker functions using less than 16 registers to prevent compiler spills to the stack during register allocation. 3) Don't pass arguments by reference to the worker function. Dereferencing a pointer to a variable inside the caller's stack frame will force a migration back to the home nodelet. Pointers to replicated data can be often be used to circumvent this rule.

Thread spawn limits: Emu's implementation of the Cilk syntax creates a lightweight thread for each independent unit of parallel work. A straightforward implementation of BFS will spawn a large number of threads dynamically. As these threads traverse the graph they migrate throughout the machine, more frequently visiting the nodelets that contain high-degree vertices. When many threads simultaneously migrate to the same nodelet, the resulting hotspot reduces performance and (on the current prototype) leads to hardware crashes. The implementation of BFS in Section 5.2 works around this issue by spawning a fixed number of threads at each nodelet and issues remote write operations to avoid migrating. Unfortunately this static work partitioning leads to additional load imbalance within a single nodelet. In Figure 5.10, a handful of threads on nodelet 0 are assigned to several high-degree vertices of an RMAT graph, delaying the exploration of the first frontier. In contrast, the thread distribution when exploring an Erdös - Rényi graph (Figure 5.11) is balanced due to the uniform degree distribution. Future versions of the Emu hardware will employ a hardware-supported credit system to control the overall amount of dynamic parallelism. This improvement will benefit BFS as well as SpMV, where the number of elements per row can vary greatly.

Workload Balance and Distributed Data Structures: One of the main challenges in

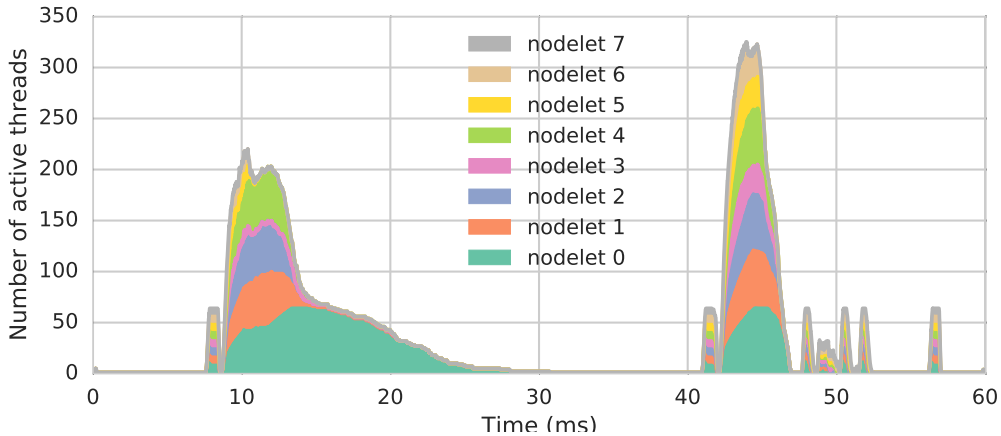


Figure 5.10: BFS benchmark thread distribution for unbalanced (RMAT) graph

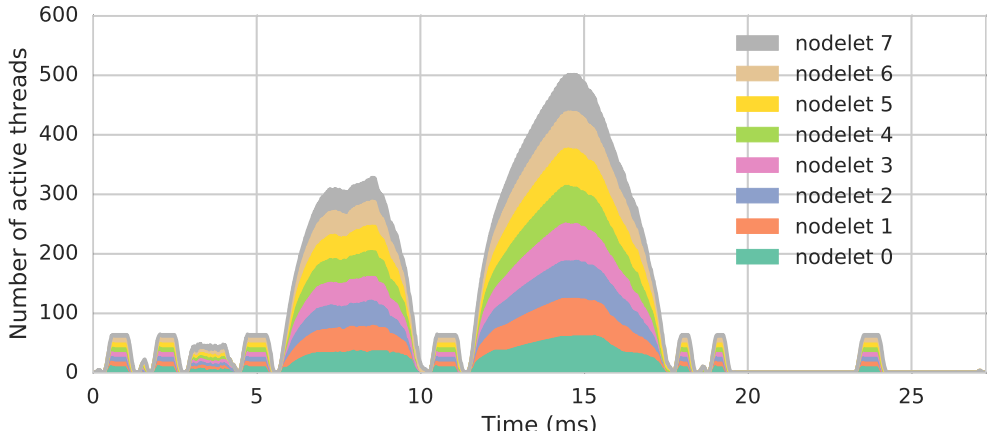


Figure 5.11: BFS benchmark thread distribution for balanced (Erdös - Rényi) graph

obtaining good performance on the Emu Chick prototype is the initial placement of data and distribution to remote nodelets. This choice of placement is critical to avoid thread migration hotspots (e.g., if all the data is placed on nodelet 0). The current limitations on dynamic parallelism make it difficult to implement an effective dynamic graph partitioning scheme, but these techniques will be essential to fully scale across across a rack-scale system.

Non-uniform PGAS Operations: Emu's implementation of PGAS utilizes "put"-style remote operations (add, min, max, etc.) and "get" operations where a thread is migrated to read a local piece of data. Thread migration is efficient when many get operations need to access the same nodelet-local memory channel. The performance difference observed between put and get operations is due to how these two operations interact differently with load balancing. A put can be done without changing the location of the thread, while a get means that multiple threads may have to share significant resources on the same nodelet for a while. Additionally, a stream of gets with spatial locality can be faster than multiple put operations. This non-uniformity means that kernels that need to access finely grained data in random order should be implemented as put operations wherever possible while get operations should only be used when larger chunks of data are read together.

5.6 Conclusion

The experiments in this work constitute the first evaluation of streaming graph algorithms on the Emu Chick hardware. Several lessons on programming the Emu system have been learned from the efforts to optimize these types of algorithms.

The Emu architecture inverts the traditional scheme of hauling data to and from a grid of processing elements. In this architecture, the data is static, and small logical units of computation move throughout the system. The load balancing is closely related to data distribution, since threads can only run on local processing elements. Performance was strongly related to how evenly the data was distributed across the nodelets of the system.

While thread migration is automatic, the programmer must be aware of the data layout in order to achieve peak performance. In one case, adopting a "put-only" mindset for random access kernels was a superior strategy when compared with simply allowing threads to migrate automatically. Finally, it was discovered that the Emu threading model is not as simple as a traditional Cilk runtime, especially with respect to dynamic parallelism. In order to saturate system performance, the programmer must be careful to orchestrate both the location and the quantity of thread spawns. While some of these issues relate only to the current prototype hardware, avoiding thread hotspots and spurious migrations due to stack accesses will continue to be performance-critical optimizations as the system matures.

CHAPTER 6
CONCLUSION AND FUTURE WORK

This work has made several contributions towards the design and implementation of a computer that is optimized for the rapid construction, modification, and analysis of massive graph datasets. Chapter 2 proposed DynoGraph, a benchmark suite for streaming graph analytics. DynoGraph measures the performance of streaming algorithms interleaved with incremental graph construction, putting focus on dynamic graph data structures that must provide efficient traversal while undergoing frequent modification. DynoGraph's real-world streaming graph inputs contain duplicates and bursts of updates to high-degree vertices, allowing system designers to plan for worst-case scenarios in load balancing and algorithm performance. As researchers develop new memory-centric architectures for irregular memory applications, including DynoGraph in the evaluation will ensure that the unique characteristics of streaming graph analytics are considered and integrated into the design.

Several techniques for improving the current state of streaming graph analytics were explored in chapter 3. In addition to a major improvement to the algorithm for parallel edge insertion in the STINGER graph engine, two proposals for near-memory graph-specific accelerators were evaluated. It was found that an accelerator can reduce the latency penalty of the pointer dereferences inherent in the traversal of a STINGER adjacency list. It was also determined that multi-channel DRAM devices do not deliver satisfactory improvements in performance when presented with a large number of small, unordered memory accesses.

Chapter 4 introduced the Emu architecture, which shows great promise for accelerating irregular applications, especially streaming graph traversal. The Emu Chick eschews deep cache hierarchies and wide memory buses in favor of a larger number of narrow memory channels, enabling high bandwidth utilization without undue reliance on spatial or temporal

locality. Initial experiments showed that selecting the correct data layout and thread spawn patterns were crucial to obtaining the best performance on this architecture. While the absolute performance of this hardware prototype is so far unimpressive compared with traditional architectures, extrapolations from results on the current hardware and simulator project that Emu will excel on irregular workloads like pointer chasing.

The investigation into the Emu architecture culminated in the design and evaluation of a custom streaming graph analytics software suite in chapter 5. Lessons learned from the initial characterization of the system were applied to create a custom data layout and parallelization strategy. It was discovered that the capability of the Emu architecture to perform remote writes to a single 64-bit integer anywhere on the system was key to efficiently implementing a scalable breadth-first search. In the course of this project, numerous recommendations and best practices were discovered, relevant both to achieving good performance on the current prototype, and to revising the design for future iterations of the hardware platform.

6.1 Future Work

The continued development of the Emu architecture presents many opportunities for future work.

It was shown that balancing the distribution of data on Emu is essential to balancing the work across the entire system. In the case of streaming graph analytics, decisions for data distribution must be made at runtime, balancing the time to construct the graph with the expected efficiency of later graph traversals. Algorithms for dynamic graph partitioning will be key to preventing "hot spots" and finding coarse-grained spatial locality as the graph is constructed on-the-fly.

HPC software engineers have come to prefer flat data structures for representing large in-memory datasets over linked data structures such as linked lists, binary trees, etc. Flat arrays often outperform linked lists due to the additional irregular memory accesses, even

for operations such as insertion and deletion, which have theoretically poor asymptotic run times [106]. Emu's narrow memory access channel, combined with its lack of cache means that contiguous, sequential accesses are no longer required to achieve maximum performance. This fact should be exploited in data structure design for Emu.

The Emu architecture currently lacks robust support for floating point operations. There are no remote atomics for floating point types, and the GC's are not designed to sustain a high rate of floating point operations per second. While the focus of Emu is sparse data sets and pointer chasing, there are some irregular applications like PageRank and machine learning that would benefit from improved floating point performance. Adding wide-word/SIMD instructions and registers to Emu would improve the situation, but would also bloat the size of a migrating thread context. Clever solutions to this problem would allow the Emu architecture to excel at running irregular floating-point algorithms.

Finally, many extensions could be proposed to the Emu remote atomic instructions. A new instruction to push an integer into a remote queue would be extremely useful, not only for graph algorithms such as BFS, but also for managing producer-consumer communication patterns between threads. It may even be possible to define compound atomics, chaining multiple remote atomic operations together into a single packet that could run at a remote memory controller. For example, a remote compare-and-swap (does this vertex already have a parent?) combined with a queue-append (push to the queue to explore in the next frontier) would be sufficient to implement the frontier exploration step of a breadth-first search.

REFERENCES

[1] O. Tange, "GNU parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb. 2011.

[2] F. Pérez and B. E. Granger, "IPython: A system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, May 2007.

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[4] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[5] M. Technology. (2015). Speed vs. latency: Why CAS latency isn't an accurate measure of memory performance, (visited on 04/12/2018).

[6] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006.

[7] C. Bienia, "Benchmarking modern multiprocessors," PhD thesis, Princeton University, Jan. 2011.

[8] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, ACM, vol. 23, Jun. 1995, pp. 24–36.

[9] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2012, pp. 235–246.

[10] K. Anderson and S. Plimpton. (2013). Firehose streaming benchmarks, (visited on 05/14/2018).

[11] D Ediger, K Jiang, J Riedy, and D. A. Bader, "Massive streaming data analytics: a case study with clustering coefficients," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–8.

[12] P. M. Kogge, "Graph analytics: Complexity, scalability, and architectures," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 1039–1047.

[13] M. Radulovic, S. A. Mckee, and E. Ayguadé, "Another trip to the wall : how much will stacked dram benefit hpc ?," pp. 0–5, 2015.

[14] D. P. Scott Beamer, Krste Asanović, "Locality exists in graph processing : workload characterization on an Ivy Bridge server," *2015 IEEE International Symposium on Workload Characterization*, 2015.

[15] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[16] R. Vuduc, O. Green, M. Dukhan, and R. Vuduc, "Branch-avoiding graph algorithms," *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pp. 212–223, 2015.

[17] R Geisberger, P Sanders, and D Schultes, "Better approximation of betweenness centrality," in *2008 Proceedings of the Tenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, ch. 8, pp. 90–100.

[18] J Riedy, "Updating PageRank for streaming graphs," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 877–884.

[19] S. Sallinen, K. Iwabuchi, S. Poudel, M. Gokhale, M. Ripeanu, and R. Pearce, "Graph colouring as a challenge problem for dynamic graph processing on distributed systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 30:1–30:12, ISBN: 978-1-4673-8815-3.

[20] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010*, no. Dm, 2010.

[21] T Dysart, P Kogge, M Deneroff, E Bovell, P Briggs, J Brockman, K Jacobsen, Y Juan, S Kuntz, R Lethin, J McMahon, C Pawar, M Perrigo, S Rucker, J Ruttenberg, M Ruttenberg, and S Stein, "Highly scalable near memory processing with migrating threads on the Emu system architecture," in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, Nov. 2016, pp. 2–9.

[22] The Hybrid Memory Cube Consortium, *Hybrid memory cube specification 2.0*, 2013.

[23] J. Kim and Y. Kim, "HBM: memory solution for bandwidth-hungry processors," in *2014 IEEE Hot Chips 26 Symposium (HCS)*, Aug. 2014, pp. 1–24.

[24] A. Suresh, P. Cicotti, and L. Carrington, "Evaluation of emerging memory technologies for HPC, data intensive applications," pp. 239–247, 2014.

[25] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2015, pp. 105–117.

[26] M Gao, G Ayers, and C Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 113–124, Oct. 2015.

[27] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," *2013 IEEE International 3D Systems Integration Conference (3DIC)*, pp. 1–7, 2013.

[28] L. Nai and H. Kim, "Instruction offloading with HMC 2.0 standard," *Proceedings of the 2015 International Symposium on Memory Systems - MEMSYS '15*, pp. 258–261, 2015.

[29] Y. Guo, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke, "Benchmarking graph-processing platforms : a vision," *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pp. 289–292, 2014.

[30] P. Konecny, "Introducing the Cray XMT," *Proc. Cray User Group meeting (CUG 2007)*, 2007.

[31] X. Wang, J. D. Leidel, and Y. Chen, "Concurrent dynamic memory coalescing on goblincore-64 architecture," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16, New York, NY, USA: ACM, 2016, pp. 177–187, ISBN: 978-1-4503-4305-3.

[32] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, and R. Lethin, "Highly scalable near memory processing with migrating threads on the Emu system architecture," in *Irregular Applications: Architecture and Algorithms (IA3), Workshop on*, IEEE, 2016, pp. 2–9.

[33] E. Bakshy, J. M. Hofman, W. A. Mason, and D. J. Watts, "Everyone's an influencer: quantifying influence on Twitter," in *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, ser. WSDM '11, New York, NY, USA: ACM, 2011, pp. 65–74, ISBN: 978-1-4503-0493-1.

[34] G. Chin Jr., S. Choudhury, J. Feo, and L. Holder, "Predicting and detecting emerging cyberattack patterns using StreamWorks," in *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, ser. CISR '14, Oak Ridge, Tennessee, USA: ACM, 2014, pp. 93–96, ISBN: 978-1-4503-2812-8.

[35] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin, " GraphBIG: understanding graph computing in the context of industrial solutions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, New York, NY, USA: ACM, 2015, 69:1–69:12, ISBN: 978-1-4503-3723-6.

[36] I. Tanase, Y. Xia, L. Nai, Y. Liu, W. Tan, J. Crawford, and C.-Y. Lin, "A highly efficient runtime and graph library for large scale graph analytics," in *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, ser. GRADES'14, Snowbird, UT, USA: ACM, 2014, 10:1–10:6, ISBN: 978-1-4503-2982-8.

[37] M. Kulkarni, M. Burtscher, C Cascaval, K. Pingali, C. Casçaval, K. Pingali, C Cascaval, and K. Pingali, "Lonestar: a suite of parallel irregular programs," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, Apr. 2009, pp. 65–76.

[38] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "CRONO: a benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*, Oct. 2015, pp. 44–55.

[39] J. Shun, G. E. Blelloch, R Geisberger, P Sanders, and D Schultes, "Ligra: a lightweight graph processing framework for shared memory," *PPoPP*, pp. 135–146, 2013.

[40] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: parallel processing of compressed graphs with Ligra+," *Data Compression Conference Proceedings*, vol. 2015-July, pp. 403–412, 2015.

[41] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015. arXiv: 1508.03619.

[42] J. Leskovec and A. Krevl. (Jun. 2014). SNAP Datasets: Stanford large network dataset collection, (visited on 05/14/2018).

[43] J. Kunegis, "Konect: The koblenz network collection," in *Proceedings of the 22Nd International Conference on World Wide Web*, ser. WWW '13 Companion, Rio de Janeiro, Brazil: ACM, 2013, pp. 1343–1350, ISBN: 978-1-4503-2038-2.

[44] T. P. Peixoto, "The graph-tool python library," *Figshare*, 2014.

[45] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group (CUG)*, 2010.

[46] M. Capotă, T. Hegeman, A. Iosup, A. Prat-Pérez, O. Erling, and P. Boncz, "Graphalytics: A big data benchmark for graph-processing platforms," GRADES'15, 7:1–7:6, 2015.

[47] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: a resilient distributed graph system on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13, New York, New York: ACM, 2013, 2:1–2:6, ISBN: 978-1-4503-2188-4.

[48] J. Webber, "A programmatic introduction to neo4j," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12, Tucson, Arizona, USA: ACM, 2012, pp. 217–218, ISBN: 978-1-4503-1563-0.

[49] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph : taking the pulse of a fast-changing and connected world," *Proceedings of the 7th ACM european conference on Computer Systems EuroSys 12*, pp. 85–98, 2012.

[50] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger, "The stapl parallel graph library," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7760 LNCS, pp. 46–60, 2013.

[51] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 40, 2012, pp. 37–48.

[52] C. X. Yi, J. Sanders, J. Nelson, B. Holt, B. Myers, L. Ceze, M. Oskin, S. Kahan, H. Chafi, and T. Harris. (2015). Graphbench, (visited on 04/12/2018).

[53] D. Gregor and A. Lumsdaine, "The parallel BGL: a generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, pp. 1–18, 2005.

[54] N. K. Ahmed, N. Duffield, T. Willke, and R. A. Rossi, "On sampling from massive graph streams," 2017. arXiv: 1703.02625.

[55] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in*

Algorithms and Architectures, ser. SPAA '12, New York, NY, USA: ACM, 2012, pp. 68–70, ISBN: 978-1-4503-1213-4.

[56] Y. Shiloach and U. Vishkin, "An O (log n) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57–67, Mar. 1982.

[57] L. L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: bringing order to the web," *World Wide Web Internet And Web Information Systems*, vol. 54, no. 1999-66, pp. 1–17, 1998. arXiv: 1111.4503v1.

[58] D. Campbell, D. Ediger, J. Poovey, and T. Goodyear, *Real-time traffic classification & graph analytics for SCinet*, Presented at the Innovating the Network for Data Intensive Science Workshop, 2014.

[59] T. Goodyear, E. Stuart, M. Fields, E. Hein, and M. Wisneski, "Identifying malicious entities through massive graph analysis of dns resolution patterns," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16, 2016.

[60] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: a recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, pp. 442–446.

[61] D. Bader, J. Berry, A. Amos-Binks, D. Chavarria-Miranda, C. Hastings, K. Madduri, and S. Poulos, "STINGER: spatio-temporal interaction networks and graphs (STING) extensible representation," *Technical Report*, pp. 1–7, 2009.

[62] L. Dagum and R. Menon, "OpenMP: an industry-standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998.

[63] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A performance evaluation of open source graph databases," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. The 1st Workshop on Parallel Programming for Analytics Applications (PPAA 2014), New York, NY, USA: ACM, 2014, pp. 11–18, ISBN: 978-1-4503-2654-4.

[64] S Beamer, A Buluç, K Asanovic, and D Patterson, "Distributed memory breadth-first search revisited: enabling bottom-up search," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, May 2013, pp. 1618–1627.

[65] D Ediger, R McColl, J Riedy, and D. A. Bader, "STINGER: high performance data structure for streaming graphs," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, ser. The IEEE High Performance Extreme Computing Conference (HPEC), Sep. 2012, pp. 1–5.

[66] G. Feng, "DISTINGER : a distributed graph data structure for massive dynamic graph processing," pp. 1814–1822, 2015.

[67] O. Green and D. A. Bader, "cuSTINGER: supporting dynamic graph algorithms for gpus," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, 2016, pp. 1–6.

[68] R. Thankachan, E. Hein, B. Swenson, and J. Fairbanks, "Integrating productivity-oriented programming languages with high-performance data structures," in *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017*, 2017, ISBN: 9781538634721.

[69] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 121–133, 1999.

[70] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *IEEE Micro*, vol. 25, no. 1, pp. 90–97, Jan. 2005.

[71] M. Karlsson, F. Dahlgren, and P. Stenstrom, "A prefetching technique for irregular accesses to linked data structures," in *Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550)*, IEEE Comput. Soc, 2000, pp. 206–217, ISBN: 0-7695-0550-3.

[72] Y. Solihin, J. Torrellas, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *Proceedings 29th Annual International Symposium on Computer Architecture*, IEEE Comput. Soc, 2002, pp. 171–182, ISBN: 0-7695-1605-X.

[73] R. Cooksey, S. Jourdan, and D. Grunwald, "A stateless, content-directed data prefetching mechanism," *ACM SIGPLAN Notices*, vol. 37, no. 10, p. 279, Oct. 2002.

[74] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a smarter memory controller," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999, pp. 70–79.

[75] X. Yu, C. J. Hughes, N. Satish, S. Devadas, S. D. Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP : indirect memory prefetcher," *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pp. 178–190, 2015.

[76] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers: Accelerating index traversals for in-memory databases," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser.

MICRO-46, Davis, California: ACM, 2013, pp. 468–479, ISBN: 978-1-4503-2638-4.

[77] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[78] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, "SQRL: hardware accelerator for collecting software data structures," *Workshop on Near Data Processing*, 2013.

[79] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[80] M. Technology. (2015). SB-850 supercomputer with hybrid memory cube, (visited on 04/12/2018).

[81] S. Lloyd, C. Hajas, M. Gokhale, S. Lloyd, and C. Hajas, "Near memory data structure rearrangement," *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pp. 283–290, 2015.

[82] H. S. Stone, "A logic-in-memory computer," *Computers, IEEE Transactions on*, no. January, pp. 73–78, 1970.

[83] V. Seshadri, M. a. Kozuch, T. C. Mowry, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, and P. B. Gibbons, "Rowclone," *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-46*, pp. 185–197, 2013.

[84] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, "NDA: near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 283–295.

[85] Z. Guz, M. Awasthi, V. Balakrishnan, M. Ghosh, A. Shayesteh, T. Suri, and S. Semiconductor, "Real-time analytics as the killer application for processing-in-memory," *Near Data Processing (WoNDP)*, pp. 10–12, 2014.

[86] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[87] T Chen, C Cher, H. M. Jacobson, J. H. Moreno, and D. A. Prener, "Active memory cube : a processing-in-memory architecture for exascale systems," vol. 59, no. 2, pp. 1–14, 2015.

[88] C. D. Kersey, S. Yalamanchili, and H. Kim, "SIMT-based logic layers for stacked DRAM architectures," *Proceedings of the 2015 International Symposium on Memory Systems - MEMSYS '15*, pp. 29–30, 2015.

[89] Q. Guo, T.-M. Low, N. Alachiotis, B. Akin, L. Pileggi, J. C. Hoe, and F. Franchetti, "Enabling portable energy efficiency with memory accelerated library," *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*, pp. 750–761, 2015.

[90] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki, "M7: Oracle's next-generation sparc processor," *IEEE Micro*, vol. 35, no. 2, pp. 36–45, Mar. 2015.

[91] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi, "The HPC Challenge (HPCC) benchmark suite," *Proceedings of the 2006 ACM/IEEE conference on Supercomputing - SC '06*, 2006.

[92] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights landing: second-generation Intel Xeon Phi product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016.

[93] C. E. Leiserson, "Programming irregular parallel applications in Cilk," in *International Symposium on Solving Irregularly Structured Problems in Parallel*, Springer, 1997, pp. 61–71.

[94] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel programming with migratable objects: Charm++ in practice," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2014, pp. 647–658.

[95] D. Ediger and D. A. Bader, "Investigating graph algorithms in the BSP model on the Cray XMT," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 1638–1645, ISBN: 978-0-7695-4979-8.

[96] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: an API for programming with millions of lightweight threads," *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, Apr. 2008.

[97] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," *Proceedings of the International Symposium on Microarchitecture*, no. i, pp. 228–241, 2015.

[98] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures - SPAA '09*, p. 79, 2009.

[99] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[100] A. D. Robinson. (2014). A parallel stable sort using C++11 for TBB, Cilk Plus, and OpenMP, (visited on 04/13/2018).

[101] M. Minutoli, S. K. Kuntz, A. Tumeo, and P. M. Kogge, "Implementing radix sort on Emu 1," *Workshop on Near-Data Processing (WoNDP)*, pp. 1–6, 2015.

[102] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy, "An initial characterization of the Emu Chick," in *In the The Eighth International Workshop on Accelerators and Hybrid Exascale Systems (ASHES), Vancouver, Canada*, 2018.

[103] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on BlueGene/L," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov. 2005, pp. 25–25.

[104] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[105] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned," in *European Conference on Parallel Processing*, Springer, 2015, pp. 438–450.

[106] B. Stroustrup, *C++11 style*, GoingNative, 2012.